

Vibe Handbook

Vibe Handbook

Índice

Git y GitHub	5
Qué es el control de versiones (y por qué lo necesitas)	6
Repositorios, commits e historial	6
Ramas: probar cosas sin miedo	7
Ver qué cambió realmente (diff)	8
El botón de “ay, no”: undo, revert, reset	8
Remotos y GitHub: tu proyecto en la nube	8
Pull requests e issues	9
.gitignore y no publicar secretos jamás	9
Tu primer repositorio, de principio a fin	10
Dejar que la IA conduzca (mientras tú sigues al mando)	10
Recap and Practice	10
La línea de comandos	12
Qué son en realidad la terminal y el shell	12
Leer el prompt	13
Los comandos de cada día	13
Ejecutar un programa y un servidor de desarrollo	14
Leer la salida y los códigos de salida	14
Detener un proceso con Ctrl-C	15
PATH y “command not found”	15
Ejecutar comandos que te da la IA — de forma segura	15
Gestores de paquetes que encontrarás	16
Repaso y práctica	16
Cómo funciona la web	17
Los dos personajes: cliente y servidor	17
Qué es en realidad una URL	18
El ciclo de solicitud y respuesta	18
Métodos HTTP: los verbos	19
Códigos de estado: ¿funcionó?	19
Encabezados y cookies	20
HTTPS y el candado	20
De un nombre a un servidor: DNS	20
Cómo se relaciona esto con tu app	21
Repaso y práctica	21
APIs, JSON y datos	22
Qué es realmente una API	23
Endpoints y REST, en términos sencillos	23
Qué es JSON	24
Los tipos de datos que encontrarás	24

Una solicitud y respuesta reales	25
Claves de API y encabezados de autenticación	25
Límites de tasa y la trampa de la factura sorpresa	26
Cómo leer la documentación de una API	26
Dejar que la IA haga la llamada, mientras tú verificas la forma	27
Resumen y práctica	27
Anatomía de una aplicación	28
El panorama general	29
Frontend, backend, base de datos, hosting	29
HTML, CSS y JavaScript	30
Paquetes y dependencias	30
Variables de entorno y configuración	31
La forma de un proyecto	32
Cómo fluye una solicitud	32
Resumen y práctica	33
La mentalidad del vibe coding	34
Qué es realmente el vibe coding	34
El cambio de mentalidad	35
El ciclo: intención → generar → revisar → refinar	36
Dónde brilla	37
Dónde falla	37
Trampas comunes	38
El buen criterio es la nueva habilidad central	38
Expectativas realistas	39
Repaso y práctica	39
Configura tu espacio de trabajo con IA	40
Elige un editor o asistente nativo de IA	41
Estructura tu proyecto para que la IA pueda navegarlo	41
Escribe un archivo de contexto (AGENTS.md)	42
Convierte el control de versiones en un reflejo	43
Mantén el ciclo de retroalimentación ajustado	43
Gestiona los secretos con cuidado	44
Una configuración local sensata hace seguros los cambios de la IA	45
Diagnósticos en vivo (LSP)	45
Extendiendo tu agente	46
macOS frente a Windows	46
TUI frente a GUI	47
Ingeniería de arneses (Harness Engineering)	47
Seguridad de costos: CLI frente a API	48
Resumen y práctica	49
Prompts como un ingeniero	50
Da contexto antes de dar la tarea	50
Especifica entradas, salidas y casos límite	51
Trabaja en pasos pequeños	52
Pide un plan antes del código	52
Itera sobre diffs, no reescrituras	53
Proporciona ejemplos	53
Dile a la IA qué NO hacer	54
Administra el contexto como un recurso	54
Antipatrones comunes que evitar	55
La mentalidad central	55

Resumen y práctica	56
De la idea a la especificación en minutos	57
Por qué una especificación, y no solo intuición	57
Paso 1: Aclarar el problema	58
Paso 2: Definir el alcance y los no objetivos	59
Paso 3: Escribir historias de usuario	59
Paso 4: Redactar un PRD ligero	60
Paso 5: Esbozar el modelo de datos	61
Paso 6: Dividir la especificación en tareas del tamaño justo	61
Mantén viva la especificación	62
Una trampa común: especificar soluciones, no problemas	62
La recompensa	63
Repaso y práctica	63
Eligiendo tu stack: los lenguajes de un vistazo	64
JavaScript / TypeScript	65
Python	65
Go	66
Rust	66
Una palabra rápida sobre los demás	67
Comparación de un vistazo	68
El costo real de una elección exótica	68
Elige según tu objetivo	69
Repaso y práctica	70
Construir funciones, vibe a vibe	71
El ciclo de construcción	71
Qué significa “rebanada vertical”	72
Un ejemplo resuelto: una función de “Favoritos”	73
Leer un diff sin entender cada línea	75
Cuándo dejar que la IA avance libremente y cuándo dirigirla manualmente	75
Mantén la aplicación siempre funcionando	75
Confirmar cambios en incrementos pequeños	76
Cuando una rebanada es demasiado grande	76
La conclusión	76
Repaso y práctica	77
Depuración con IA	78
La mentalidad central	78
Una lista de verificación para depurar	79
Alimentar a la IA con el contexto correcto	80
Leer errores que no entiendes	81
Hipótesis, registros y búsqueda binaria	81
Causa raíz frente a síntoma	82
Cuando la IA da vueltas en círculo	83
Entender el arreglo antes de conservarlo	83
La conclusión	83
Resumen y práctica	83
Infraestructura de servidores	85
Los cinco lugares donde puede vivir el código	85
Un par de términos de jerga, descifrados	86
La tabla de compensaciones	87
Cuándo elegir cuál	87

Cómo la IA cablea los despliegues y la configuración	88
Una ruta por defecto recomendada para el constructor solitario	89
Qué significa realmente “gestionado” (y qué sigue siendo tuyo)	90
Una lista de verificación rápida antes del lanzamiento	90
La conclusión	91
Repaso y práctica	91
Infraestructura de bases de datos	92
Las tres formas de almacenar datos	93
Gestionado vs autoalojado	93
Fundamentos del diseño de esquemas	94
Migraciones: cambiar el esquema con seguridad	95
Indexación	96
Caché	97
Colas	97
Errores comunes de principiante	97
Dirigir a la IA para construir tu capa de datos	98
Recapitulación y práctica	98
Barreras de protección, pruebas y revisiones	99
Por qué necesitas barreras de protección al dirigir a la IA	100
Pruebas: tu red de seguridad	100
Comprobación de tipos y linting: errores gratis detectados	101
Delimitar el alcance de los cambios de la IA	102
Leer diffs con espíritu crítico	102
Puertas de revisión: una lista de comprobación antes de fusionar	103
Fundamentos de CI: haz que las comprobaciones sean automáticas	103
Hábitos seguros por defecto	104
Cuando una barrera de protección grita: haz triaje antes de arreglar	104
Repaso y práctica	105
Publicar en producción	106
Elige un host y despliega	107
Dominios personalizados y DNS	108
Variables de entorno y secretos	109
Conectando los pagos	109
Staging y reversiones	110
Monitoreo y seguimiento de errores	111
Rendimiento y caché	111
Lista de verificación previa al lanzamiento	112
Después de lanzar	112
Repaso y práctica	112
Casos de estudio: de cero al lanzamiento	114
Caso de estudio 1: Una página de aterrizaje multilingüe con pagos	114
Caso de estudio 2: Una pequeña herramienta SaaS con autenticación y base de datos	116
Caso de estudio 3: Una utilidad de automatización personal	118
Caso de estudio 4: Un sitio de contenido que no tienes que atender	120
Qué tienen en común los cuatro	122
Recapitulación y práctica	122
Seguridad para vibe coders	124
Por qué el código generado por IA es inseguro por defecto	124
Inyección: cuando la entrada se convierte en código	125
Secretos y claves de API: la trampa de la clave expuesta	126

Autenticación vs. autorización: el endpoint que cualquiera puede llamar	127
Manejo de archivos y subidas	128
Riesgo de dependencias: el paquete que la IA se inventó	128
El filtro de revisión de seguridad	128
Complacencia con la automatización, edición seguridad	129
Resumen y práctica	130
Orquestación de agentes	131
Los tres niveles	131
Cuándo dejarlo correr vs. dirigir manualmente	132
Delimitar una tarea para que no pueda desviarse	132
Agentes en segundo plano y en paralelo	133
Multiagente: expansión, adversario, síntesis	134
Revisar salida a gran escala	135
Costo y bucles descontrolados	136
Límites honestos	136
Resumen y práctica	137

Git y GitHub



Si solo vas a aprender una herramienta antes de empezar a programar con vibe coding, que sea esta. Git es lo que te permite ser valiente. Es la razón por la que puedes decirle a una IA “reescribe todo este archivo” y no perder el sueño, porque siempre puedes volver a la versión que funcionaba.

Aquí va la versión honesta de por qué esto importa. Cuando le pides a una IA que construya software, va a romper cosas — con regularidad. Reescribirá un archivo con total confianza, borrará la parte que funcionaba, y te entregará un desastre. La diferencia entre una tarde frustrante y una semana perdida es si tenías una red de seguridad. Git es esa red. Este capítulo la enseña desde cero, sin necesidad de experiencia previa en programación.

Qué es el control de versiones (y por qué lo necesitas)

Imagina un videojuego con puntos de guardado. Juegas una sección complicada, guardas, intentas algo arriesgado. Si sale mal, cargas la última partida guardada en lugar de empezar todo el juego de nuevo. **El control de versiones son los puntos de guardado de tu proyecto.**

Sin él, tu proyecto tiene exactamente un estado: el que sea que tenga ahora mismo. Si la IA estropea tu código, la versión buena desaparece. Con control de versiones, mantienes una línea de tiempo completa de cada estado guardado, y puedes saltar de vuelta a cualquiera de ellos.

Git es la herramienta de control de versiones más común. Se ejecuta en tu computadora y registra silenciosamente instantáneas de tu proyecto cada vez que se lo pides. Algunas cosas que te da gratis:

- **Una red de seguridad** — deshacer un cambio malo incluso horas después.
- **Un historial** — ver exactamente qué cambió, cuándo y por qué.
- **Confianza** — experimentar libremente, porque nunca se pierde nada de verdad.

Ese último punto es el verdadero premio. El vibe coding funciona mejor cuando te mueves rápido y pruebas cosas. Git es lo que hace que moverse rápido sea seguro en lugar de imprudente.

Imagina el historial de tu proyecto como una fila de puntos de guardado a los que siempre puedes volver:

```
past -----> now

( C1 )-->( C2 )-->( C3 )-->( C4 )-->( C5 )
  |         |         |         |         |
"initial" "homepage" "login"  "oops, broke" "fixed it"

each ( C ) = one commit = one save point you can jump back to
```

Repositorios, commits e historial

Tres palabras que escucharás constantemente. Vamos a desmitificarlas.

Un **repositorio** (o “repo”) es simplemente una carpeta que Git está vigilando. Conviertes una carpeta común en un repositorio una vez, y a partir de entonces Git rastrea todo lo que hay dentro de ella.

Un **commit** es un punto de guardado. Es una instantánea de cada archivo en un momento dado, más un mensaje corto tuyo describiendo qué cambió (“añadida la página de inicio”, “arreglado el botón de registro que estaba roto”). Haces commits constantemente — piensa en cada uno como un punto de control al que podrías querer volver.

El **historial** es la lista ordenada de todos tus commits. Es la línea de tiempo. Puedes recorrerla, leer qué fue cada cambio, y viajar de vuelta a cualquier punto.

Así es como se ve esto en la práctica. No te preocupes por memorizarlo — normalmente dejarás que la IA ejecute estos comandos por ti — pero sigue leyendo para que las palabras dejen de ser un misterio:

```
# Turn the current folder into a Git repository (do this once)
git init

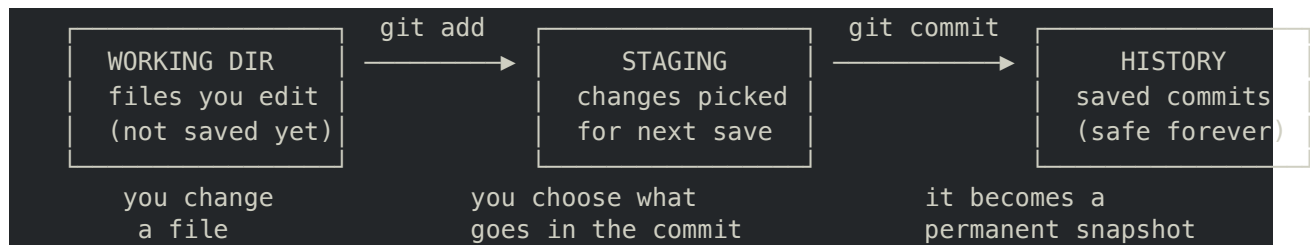
# See which files have changed since your last save point
git status
```

```
# Stage your changes, then save them as a commit with a message
git add .
git commit -m "Add the homepage layout"

# Look at your history – every commit, newest first
git log --oneline
```

`git add .` es tu forma de decir “incluye todos mis cambios en el próximo punto de guardado.” `git commit` es el guardado real. El `-m "...` es tu nota para tu yo futuro. Escribe notas que entenderías dentro de un mes.

Esos dos comandos mueven tu trabajo a través de tres etapas. Un archivo que editas no está guardado hasta que ha viajado todo el camino hacia la derecha:



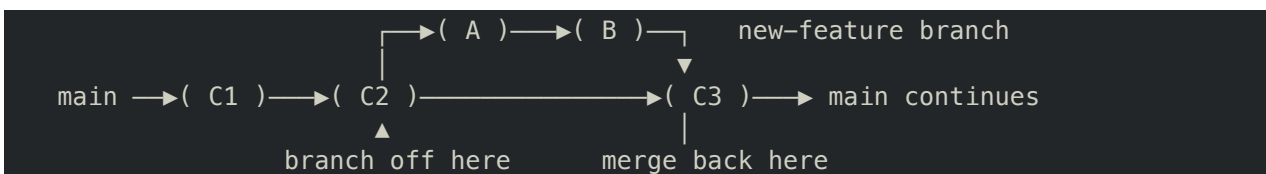
Ramas: probar cosas sin miedo

Una **rama** (branch) es una copia paralela de tu proyecto donde puedes experimentar. Tu trabajo principal vive en una rama que normalmente se llama `main`. Cuando quieres probar algo — una función nueva, una reescritura arriesgada de la IA — creas una rama nueva, haces el trabajo ahí, y tu `main` permanece intacta y segura.

Si el experimento funciona, lo fusionas (merge) de vuelta a `main`. Si no funciona, descartas la rama y `main` nunca supo que ocurrió.

Así es exactamente como deberías tratar los cambios grandes de la IA. “Construye todo el flujo de pagos” merece su propia rama. Si la IA produce algo bueno, lo fusionas. Si produce un desastre, borras la rama y no has perdido nada.

Visualmente, una rama se separa de `main`, crece por su cuenta, y luego se fusiona de vuelta:



```
# Create a new branch and switch to it
git checkout -b new-feature

# ...do your work, make commits...

# Switch back to your safe main branch
git checkout main

# Merge the good work from your branch into main
git merge new-feature
```

Ver qué cambió realmente (diff)

Antes de guardar un cambio — y *especialmente* antes de aceptar un cambio que hizo una IA — quieres ver exactamente qué es diferente. Eso es un **diff**. Te muestra, línea por línea, qué se añadió (normalmente en verde) y qué se eliminó (normalmente en rojo).

Este es uno de tus hábitos más importantes. Toda la filosofía del vibe coding es *entender lo que publicas*. No tienes que escribir el código, pero sí tienes que mirar qué cambió y entenderlo a grandes rasgos. El diff es donde haces eso.

```
# See what you've changed but not yet committed
git diff

# See what's staged and ready to be committed
git diff --staged
```

Cuando la IA dice “listo, actualicé el archivo,” tu siguiente paso es mirar el diff. Si ves que borró algo que necesitabas, lo detectas *ahora* — no después de haberlo publicado.

El botón de “ay, no”: undo, revert, reset

Esta es la parte que convierte a Git de una tarea molesta en un superpoder. Cuando algo sale mal — y saldrá mal — así es como lo deshaces.

- **Descartar cambios no guardados** — editaste (o la IA editó) algunos archivos y las cosas empeoraron, pero aún no has hecho commit. Puedes descartar todo y volver a tu último commit bueno.
- **Revertir un commit** — guardaste un cambio, y luego te diste cuenta de que era malo. `git revert` crea un commit *nuevo* que deshace el malo, manteniendo tu historial honesto.
- **Retroceder a un punto anterior** — quieres rebobinar tu proyecto a como estaba hace varios commits.

```
# Undo all uncommitted changes and go back to your last commit
git restore .

# Safely undo a specific bad commit (creates a new "undo" commit)
git revert <commit-id>

# Rewind to an earlier commit (powerful — ask the AI to explain before using)
git reset --hard <commit-id>
```

Una advertencia: `git reset --hard` descarta trabajo de forma permanente. Es la herramienta correcta a veces, pero es con la que hay que tener cuidado. Cuando tengas dudas, pregúntale a la IA “¿qué le hará exactamente este comando a mis archivos?” antes de ejecutarlo.

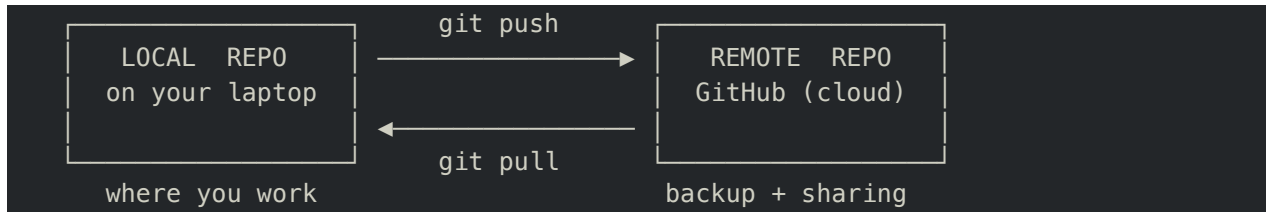
Remotos y GitHub: tu proyecto en la nube

Todo lo que hemos visto hasta ahora ocurre en tu propia computadora. **GitHub** es un sitio web que almacena una copia de tu repositorio en la nube. Esa copia se llama un **remoto** (remote).

¿Por qué molestarse? Tres razones: es un respaldo si tu laptop muere, es cómo compartes código con otras personas (o con herramientas de IA y servicios de despliegue), y es donde ocurre la colaboración.

Conectas tu repositorio local a un remoto una vez, y luego haces **push** (enviar tus commits a GitHub) y **pull** (traer commits desde GitHub).

Piensa en dos copias de tu proyecto — una en tu laptop, otra en la nube — mantenidas sincronizadas mediante push y pull:



```
# Connect your local repo to a GitHub repo (one time)
git remote add origin https://github.com/yourname/yourproject.git

# Send your commits up to GitHub
git push origin main

# Bring down any commits that are on GitHub but not on your computer
git pull origin main
```

Piensa en push como “guardar en la nube” y en pull como “traer lo más reciente de la nube.”

Pull requests e issues

Dos funciones de GitHub que conocerás rápidamente.

Un **pull request** (PR, solicitud de extracción) es una propuesta para fusionar una rama en otra, con un espacio integrado para revisar los cambios antes de que se incorporen. Incluso trabajando en solitario, los PR son útiles: te dan una pantalla limpia que muestra cada cambio, para que puedas revisar el trabajo de la IA antes de fusionarlo en main. Es el hábito del diff, formalizado.

Un **issue** (incidencia) es una nota rastreada sobre algo que hay que hacer o arreglar — un error, la idea de una función, un pendiente. Los issues son cómo mantienes una lista de tareas pegada al propio proyecto en lugar de dispersa en tu cabeza.

.gitignore y no publicar secretos jamás

Algunos archivos *nunca* deben entrar en Git. Los principales:

- **Secretos** — claves de API (Application Programming Interface, interfaz de programación de aplicaciones), contraseñas, credenciales de bases de datos. Si haces commit de un secreto en GitHub, trátalo como filtrado, incluso si lo borras después. El historial lo recuerda.
- **Basura** — archivos temporales, dependencias descargadas (como `node_modules`), resultados de compilación. Enormes e inútiles de rastrear.

Controlas esto con un archivo llamado `.gitignore`. Enumeras patrones en él, y Git finge que esos archivos no existen. Esto es crítico para quienes hacen vibe coding, porque a las herramientas de IA les encanta crear archivos de configuración con claves reales dentro, y es fácil hacer commit de uno por accidente.

```
# A simple .gitignore file
.env
node_modules/
*.log
```

La línea `.env` es la importante — ahí es donde suelen vivir los secretos. Añade `.gitignore` *antes* de tu primer commit, y adquiere el hábito de preguntarle a la IA: “¿hay algo en este commit que no debería ser público?”

Tu primer repositorio, de principio a fin

Aquí está todo el ciclo en un solo lugar. Crea una carpeta, conviértela en repositorio, guarda tu trabajo, y publícalo en GitHub:

```
# 1. Start a new project
mkdir my-first-project
cd my-first-project
git init

# 2. Protect your secrets before anything else
echo ".env" > .gitignore

# 3. Make your first save point
git add .
git commit -m "Initial commit"

# 4. Connect to GitHub and push it up
git remote add origin https://github.com/yourname/my-first-project.git
git push -u origin main
```

Eso es todo. Ahora tienes un proyecto con una red de seguridad y un respaldo en la nube. Cada vez que hagas progreso, repite el ritmo add □ commit □ push.

Dejar que la IA conduzca (mientras tú sigues al mando)

Aquí está la parte liberadora: no tienes que memorizar ninguno de estos comandos. Puedes decirle a la IA “haz commit de esto con un mensaje sensato” o “crea una rama para la nueva función y cámbiate a ella,” y ejecutará los comandos correctos. Esa es una forma perfectamente válida de trabajar.

La trampa — y es la misma trampa que en el resto de este libro — es que sigues siendo responsable de entender *qué* hizo, incluso si tú no lo escribiste. El camino intermedio que realmente funciona:

- **Deja que la IA ejecute los comandos**, pero pídele que te diga qué hace cada uno en lenguaje sencillo las primeras veces.
- **Lee el diff tú mismo** antes de cualquier commit. Esto no es negociable. Es cómo detectas que la IA borró algo que no debía.
- **Haz commits con frecuencia, en piezas pequeñas**. Muchos puntos de guardado pequeños superan a uno gigante, porque retroceder es preciso en lugar de todo-o-nada.
- **Crea una rama antes de algo grande**. Dile a la IA que cree una rama antes de una reescritura importante, para que main permanezca segura.
- **Cuando un comando suene destructivo** (cualquier cosa con `reset --hard`, `force-push`, o borrar ramas), pregunta “¿qué hace exactamente esto?” antes de aprobarlo.

Tú eres el líder técnico. La IA es tu compañero de equipo rápido, capaz y ocasionalmente imprudente. Git es el proceso que evita que ese compañero de equipo cause daño real. Apréndelo lo suficientemente bien como para dirigirlo con confianza, y nunca más temerás el botón de “reescribir todo” — porque siempre tendrás un camino de vuelta.

Recap and Practice

Puntos clave

- Git es **puntos de guardado para tu proyecto** — una línea de tiempo completa a la que puedes saltar de vuelta, que es lo que te permite soltar a la IA sin miedo.
- El ritmo central es add □ commit □ push: preparar tus cambios, guardarlos como un commit con un mensaje claro, y enviarlos a GitHub para respaldo y compartición.

- **Crea una rama antes de algo grande, y lee el diff antes de cada commit** — así es como detectas que la IA borró algo que no debía.
- Un `.gitignore` mantiene los secretos (como `.env`) y la basura (como `node_modules/`) fuera de tu historial. Un secreto publicado en GitHub queda filtrado para siempre, incluso si lo borras después.
- Puedes dejar que la IA ejecute los comandos, pero sigues siendo responsable de entender qué hizo cada uno — especialmente los destructivos como `reset --hard`.

Pruébalo

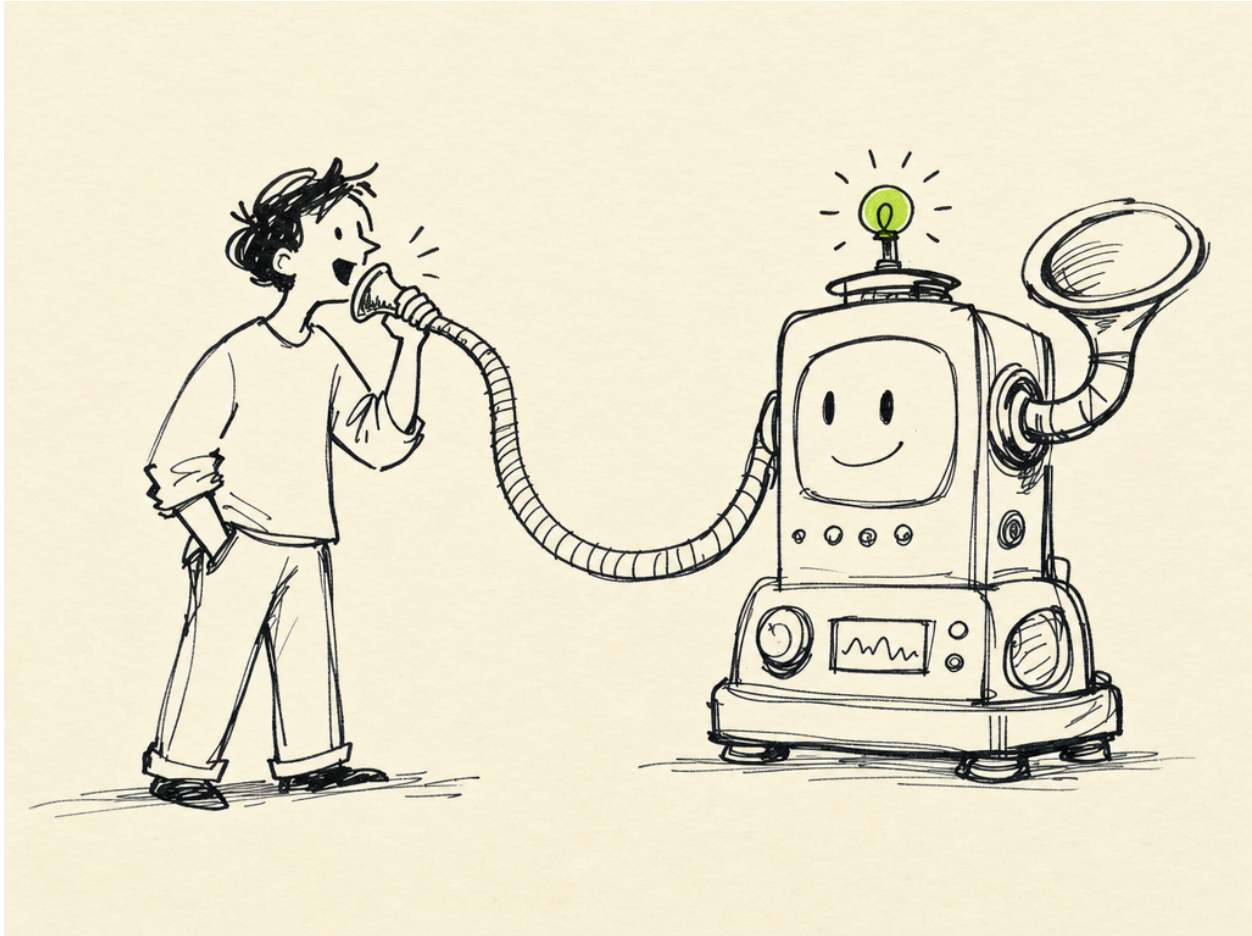
Crea una carpeta completamente nueva y ejecuta todo el ciclo tú mismo, a mano: `git init`, crea un `.gitignore` con `.env` dentro, añade un archivo sencillo `notes.txt`, y luego `git add .` y `git commit -m "Initial commit"`. Ahora ejecuta `git log --oneline` para ver tu punto de guardado, edita el archivo, y ejecuta `git diff` para ver a Git mostrarte exactamente qué cambió. Acabas de usar todos los hábitos centrales de este capítulo en un proyecto donde nada puede salir mal.

Prompt del capítulo

```
I'm learning Git as a beginner and I want you to run the commands for me,
but teach as you go. For the task below:
<describe what you want to do – e.g. "save my current work" or "try a risky change safely">

- Tell me which Git commands you'll run and what each one does in plain English.
- If anything is destructive (reset --hard, force-push, deleting a branch),
  warn me clearly and explain what I'd lose before doing it.
- Before any commit, show me the diff and summarize what changed.
- Remind me if there's anything in the change that shouldn't be public (secrets, keys).
```

La línea de comandos



La primera vez que ves a una herramienta de IA para programar hacer su trabajo, una ventana negra se llena de texto que se desplaza y un cursor parpadeante espera a que escribas. Esa ventana es la línea de comandos. Para muchas personas que no programan es la parte más aterradora de todo el recorrido: parece sacada de una película, como si un movimiento en falso fuera a borrar tu disco duro. No lo hará. La línea de comandos es simplemente otra forma de decirle a tu computadora qué hacer: con palabras en lugar de clics.

Este capítulo la desmitifica. No necesitas memorizar cien comandos. Necesitas entender qué está pasando, reconocer un puñado de comandos cotidianos y saber cómo ejecutar de forma segura los que te entregue una IA. Con eso basta para dejar de tener miedo y empezar a ser efectivo.

Qué son en realidad la terminal y el shell

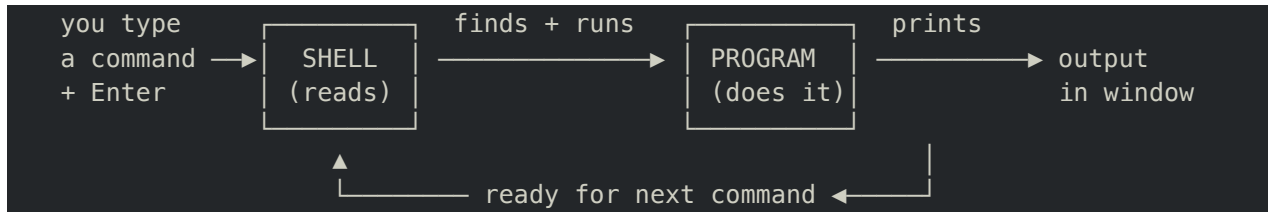
Cuando usas una aplicación normal, haces clic en botones y la aplicación traduce esos clics en instrucciones para la computadora. La línea de comandos se salta los botones. Escribes una instrucción, presionas Enter, y la computadora la ejecuta.

Dos palabras que escucharás mucho:

- **Terminal** — la ventana en la que escribes. Es solo una caja de texto con un historial. (En Mac es “Terminal” o “iTerm”; en Windows es “PowerShell” o “Windows Terminal”; en Linux normalmente es simplemente “Terminal”).

- **Shell** — el programa que está *dentro* de esa ventana y que lee lo que escribes y lo ejecuta. El más común se llama **bash**; las Mac usan uno llamado **zsh**. Se comportan de forma casi idéntica para todo lo que se ve en este libro.

Este es el ciclo, cada vez que presionas Enter: el shell lee lo que escribiste, encuentra y ejecuta el programa correspondiente, e imprime de vuelta en la ventana lo que ese programa produjo.



¿Por qué las herramientas de IA para programar viven aquí? Porque la línea de comandos es el panel de control universal del desarrollo de software. Instalar herramientas, arrancar tu aplicación, ejecutar pruebas, desplegarla en internet: todo eso sucede a través de comandos escritos. Un agente de IA funciona *escribiendo los mismos comandos que escribirías tú*, solo que más rápido y sin errores de tipeo. Cuando lo ves trabajar, estás viéndolo usar exactamente la misma puerta que tú usarás.

Leer el prompt

Antes de que escribas nada, el shell te muestra un **prompt** (símbolo de espera) — un pequeño fragmento de texto que termina en un símbolo, esperando que escribas algo. Puede verse así:

```
mini@laptop ~/projects/my-app $
```

Leyendo de izquierda a derecha, eso te dice: tu nombre de usuario (mini), tu computadora (laptop), dónde te encuentras actualmente (la carpeta ~/projects/my-app), y luego un \$ que significa “te toca a ti”. El ~ es la forma abreviada de tu carpeta de inicio (home). El símbolo al final suele ser \$ o %; a veces >. No escribas el símbolo — es solo el shell diciendo que está listo.

Esto importa más de lo que parece. La mayor parte de la confusión con la línea de comandos viene de no saber *dónde* estás. El prompt te lo dice, cada vez.

Los comandos de cada día

Aquí están los comandos que verás constantemente. Léelos como frases: un verbo, a veces seguido de un objetivo.

```

pwd          # "print working directory" – where am I right now?
ls           # list the files and folders here
ls -la       # list everything, including hidden files, with details
cd projects  # change directory – move into the "projects" folder
cd ..        # move up one folder (".." means "the folder above")
cd ~         # go back to your home folder
mkdir my-app # make a new folder called "my-app"
cp notes.txt backup.txt # copy a file (original stays, copy is made)
mv old.txt new.txt      # move or rename a file
rm draft.txt           # remove (delete) a file
  
```

Unas cuantas notas que te ahorrarán disgustos de verdad:

Tus archivos viven en un árbol de carpetas anidadas, y cd es la forma de recorrerlo. Aquí hay uno pequeño, con pwd mostrando dónde estarías parado si hubieras hecho cd hacia my-app:

```

/                               ← the top ("root")
└─ Users/
  
```

```

└─ mini/           ← your home folder (the ~ shortcut)
   └─ projects/
      ├── my-app/   ← cd here → pwd shows /Users/mini/projects/my-app
      │   ├── src/
      │   └── notes.txt
      └── old-app/

```

- **cd es cómo te mueves de un lado a otro.** Combinado con `pwd` y `ls`, es el 80 % de todo lo que harás alguna vez. Te mueves a la carpeta correcta, miras alrededor, y luego ejecutas algo.
- **Las rutas pueden ser relativas o absolutas.** `cd projects` es relativa a donde estás; `cd /Users/mini/projects` es la dirección completa desde la cima.
- **rm es permanente.** No existe una Papelera de reciclaje en la línea de comandos. Cuando haces `rm` a un archivo, desaparece — sin “¿estás seguro?”, sin deshacer. Ten especial cuidado con `rm -rf`, que elimina una carpeta y todo su contenido sin preguntar. Si en algún momento no estás seguro de qué va a borrar un comando `rm`, no lo ejecutes. Pídele a la IA que te lo explique primero.

Ejecutar un programa y un servidor de desarrollo

Construir software significa ejecutarlo. La mayoría de las veces escribirás el nombre de un programa seguido de lo que quieres que haga.

```

node app.js      # run a JavaScript file with Node
python script.py # run a Python file
npm run dev      # start the project's development server

```

Ese último es especial. Un **servidor de desarrollo** (dev server) es un programa que ejecuta tu aplicación en tu propia computadora para que puedas verla en un navegador en una dirección como `http://localhost:3000`. Lo clave para entender: **sigue en ejecución**. La terminal se verá “atascada” — deja de darte un prompt nuevo y en su lugar muestra un flujo continuo de actividad. Eso no está congelado; eso está funcionando. Tu aplicación está viva mientras ese comando siga corriendo. Abre tu navegador, visita la dirección que imprime, y verás tu aplicación.

Cuando termines, la detienes (siguiente sección).

Leer la salida y los códigos de salida

Después de que se ejecuta un comando, normalmente imprime algo. Aprender a *leer* esa salida es una habilidad real, porque es la forma en que tú y la IA descubren si las cosas funcionaron.

- **La salida normal** es solo información — archivos listados, mensajes de progreso, un servidor diciendo que arrancó.
- **Los errores** normalmente lo dicen con claridad: `Error:`, `command not found`, `permission denied`, `cannot find module`. Las palabras después de los dos puntos son tu pista. Copia el error completo y pégaselo a tu IA — es lo más útil que puedes entregarle cuando algo se rompe.

Detrás de escena, cada comando termina con un **código de salida** invisible: `0` significa éxito, cualquier otra cosa significa un problema. Rara vez necesitarás comprobarlo a mano, pero lo escucharás mencionar. Puedes ver el último así:

```

echo $?          # prints the exit code of the last command (0 = OK)

```

Si un comando falló en silencio y no estás seguro, `echo $?` te dice si realmente tuvo éxito.

Detener un proceso con Ctrl-C

Ese servidor de desarrollo sigue corriendo hasta que lo detienes. El “detener” universal es **Ctrl-C** — mantén presionada la tecla Control y presiona C. Envía una señal educada de “por favor detente ahora” a lo que sea que esté corriendo, y recuperas tu prompt.

- **Ctrl-C** — detiene el programa que se está ejecutando actualmente en esta terminal. Tu recurso de referencia para terminar un servidor de desarrollo o cancelar un comando que está tardando demasiado.
- **Cerrar la ventana de la terminal** también detiene todo lo que se ejecuta dentro de ella, pero Ctrl-C es más limpio.

Recuerda Ctrl-C y nunca más te sentirás atrapado por una terminal “atascada”.

PATH y "command not found"

Tarde o temprano escribirás un comando y obtendrás `command not found`. Esto casi nunca significa que hayas hecho algo mal; significa que la computadora no sabe dónde vive ese programa.

El shell encuentra programas revisando una lista de carpetas llamada ***PATH***. Cuando escribes npm, el shell recorre cada carpeta en PATH buscando un programa llamado npm. Si no está en ninguna de ellas, obtienes `command not found`.

```
you type: npm
          |
          ▼
  search PATH folders, in order, for "npm":
  /usr/local/bin → not here
  /usr/bin       → not here
  /opt/bin       → FOUND ✓ → run it

  if none of the folders have it → "command not found"

echo $PATH          # show the list of folders the shell searches
```

En la práctica, `command not found` normalmente significa una de dos cosas: la herramienta aún no está instalada, o se instaló en algún lugar que no está en tu `PATH`. Ambas son comunes y solucionables. Pégale el error a tu IA y dile qué estabas tratando de ejecutar — arreglar problemas de `PATH` es exactamente el tipo de trabajo de configuración tedioso en el que una IA es buena quiándote.

Ejecutar comandos que te da la IA — de forma segura

Una gran parte de la programación asistida por IA (vibe coding) es que la IA diga “ejecuta este comando” y te entregue una línea para pegar. Esto es normal y está bien. Pero adopta un hábito que te protegerá para siempre: **lee antes de ejecutar**.

No necesitas entender cada modificador (flag). Necesitas una idea aproximada de qué *hace* el comando, especialmente el verbo al principio:

- `ls`, `pwd`, `cd`, `cat`, `echo` — inofensivos. Miran cosas o se mueven de un lado a otro. Ejecútalos libremente.
- `npm install`, `pip install` — descargan e instalan herramientitas. Generalmente seguros, pero cambian tu proyecto.
- `rm`, `mv`, cualquier cosa con `sudo`, cualquier cosa que canalice una descarga directamente a tu shell (`curl ... | bash`) — **frena**. Estos pueden borrar cosas o ejecutar código con poder total sobre tu máquina.

Si un comando se ve peligroso o no reconoces el verbo, pregunta: “¿Qué hace este comando y es seguro ejecutarlo?” Una buena IA te lo explicará con claridad. Nunca pegues un comando de un sitio web al azar sin esa comprobación. La línea de comandos hace exactamente lo que se le indica — incluidas las cosas peligrosas — así que el pequeño hábito de leer primero es tu cinturón de seguridad.

Gestores de paquetes que encontrarás

El software moderno se construye a partir de miles de piezas pequeñas ya escritas. Los **gestores de paquetes** (package managers) son las herramientas que descargan y organizan esas piezas. Te encontrarás con dos constantemente:

```
npm install          # install everything this JavaScript project needs
npm install react    # add a specific package called "react"
pip install requests # install a Python package called "requests"
```

- **npm** (Node Package Manager, gestor de paquetes de Node) es el gestor de paquetes para proyectos de JavaScript y Node. `npm install` (a menudo mostrado como `npm i`) lee un archivo que enumera lo que el proyecto necesita y lo descarga todo.
- **pip** hace el mismo trabajo para Python.

A menudo ejecutarás `npm install` una vez cuando abras un proyecto por primera vez, y luego te olvidarás de él. Cuando la IA agregue una funcionalidad que necesite un paquete nuevo, te dirá que lo instales. Ahora ya sabes qué significa eso: es solo traer un bloque de construcción, no algo a lo que temer.

Eso es la línea de comandos. No es un arma de hacker — solo es una caja de texto que hace exactamente lo que le pides. Aprende a leer el prompt, reconoce una docena de comandos, detén un proceso descontrolado con `Ctrl-C`, y lee antes de ejecutar. Todo lo demás, puedes preguntarlo.

Repaso y práctica

Conclusiones clave

- La línea de comandos es solo una caja de texto que hace exactamente lo que le dices — palabras en lugar de clics. Un agente de IA funciona escribiendo los mismos comandos que escribirías tú.
- El **prompt** te dice *dónde estás* cada vez; la mayor parte de la confusión viene de no saber en qué carpeta estás actualmente. `pwd`, `ls` y `cd` son el 80 % de todo lo que harás alguna vez.
- **rm es permanente** — sin Papelera, sin deshacer. Frena con `rm`, `sudo`, y cualquier cosa que canalice una descarga hacia tu shell (`curl ... | bash`).
- Un **servidor de desarrollo** sigue corriendo y se ve “atascado” a propósito — eso es señal de que está funcionando. **Ctrl-C** detiene cualquier programa en ejecución y te devuelve tu prompt.
- **Lee antes de ejecutar.** No necesitas conocer cada modificador, solo una idea aproximada del verbo. Cuando tengas dudas, pregúntale a la IA “¿qué hace este comando y es seguro?”

Práctico

Abre tu terminal y explora sin cambiar nada: ejecuta `pwd` para ver dónde estás, `ls` para ver qué hay ahí, haz `cd` a una carpeta y `ls` de nuevo, luego `cd ..` para volver a subir. Observa cómo se actualiza el prompt cada vez. Ahora ejecuta un comando que falle a propósito — escribe `ls` (un error de tipeo) y lee el mensaje de `command not found`. Acabas de practicar los dos comandos más seguros y más usados, y aprendiste a leer un error, todo sin tocar un solo archivo.

Prompt del capítulo

```
I'm a beginner on the command line and a bit nervous about running things.
Here's a command you (or a tutorial) gave me to run:
<paste the command>

– Explain in plain English what this command does, step by step.
```

- Tell me clearly whether it's safe, or whether it could delete files, change my system, or run code with full permissions.
- If it's risky, suggest a safer way to do the same thing, or what to check first.
- Tell me what folder I should be in before I run it.

Cómo funciona la web



Antes de construir algo para la web, ayuda saber qué es la web en realidad. No necesitas un título en ciencias de la computación, y no necesitas memorizar nada. Solo necesitas un modelo mental que funcione: una imagen en tu cabeza de lo que pasa cuando escribes una dirección y aparece una página.

Este capítulo construye esa imagen. Todo lo que hace una app hecha con *vibe coding*, desde un formulario de inicio de sesión hasta un pago, es alguna versión del mismo baile sencillo: una computadora pregunta, otra computadora responde. Una vez que puedas ver ese baile con claridad, el resto del libro deja de sentirse como magia y empieza a sentirse como plomería.

Los dos personajes: cliente y servidor

Casi todo en la web es una conversación entre dos roles.

- **El cliente** es lo que *pregunta*. La mayoría de las veces es un navegador web —Chrome, Safari, Firefox— corriendo en tu teléfono o laptop. Es el cliente que llega al mostrador.

- **El servidor** es lo que *responde*. Es una computadora sentada en algún centro de datos, funcionando todo el día, esperando solicitudes. Es la cocina detrás del mostrador.

El cliente nunca hace el trabajo del servidor, y el servidor nunca hace el trabajo del cliente. El navegador pregunta: “Dame la página de inicio”. El servidor responde: “Aquí está”. Esa es toda la relación. Un servidor no es una caja mágica especial: es simplemente una computadora común cuyo trabajo es esperar preguntas y enviar respuestas.

Una palabra más que oírás constantemente: **request** (la pregunta que envía el cliente, es decir, la “solicitud”) y **response** (la respuesta que devuelve el servidor). Ten a mano esas dos palabras; son la columna vertebral de todo lo que sigue.

Qué es en realidad una URL

Una URL (Uniform Resource Locator, “localizador uniforme de recursos”) es la dirección que pides. Parece una cadena larga de texto, pero en realidad es un puñado de partes etiquetadas cosidas entre sí. Desarma esta:

`https://shop.example.com/products/shoes?size=10`

<code>https</code>	<code>scheme</code>	– how to talk (use HTTPS, the secure way)
<code>shop.example.com</code>	<code>host</code>	– which server to talk to
<code>/products/shoes</code>	<code>path</code>	– which page or thing you want from it
<code>?size=10</code>	<code>query</code>	– extra details (“the size-10 ones”)

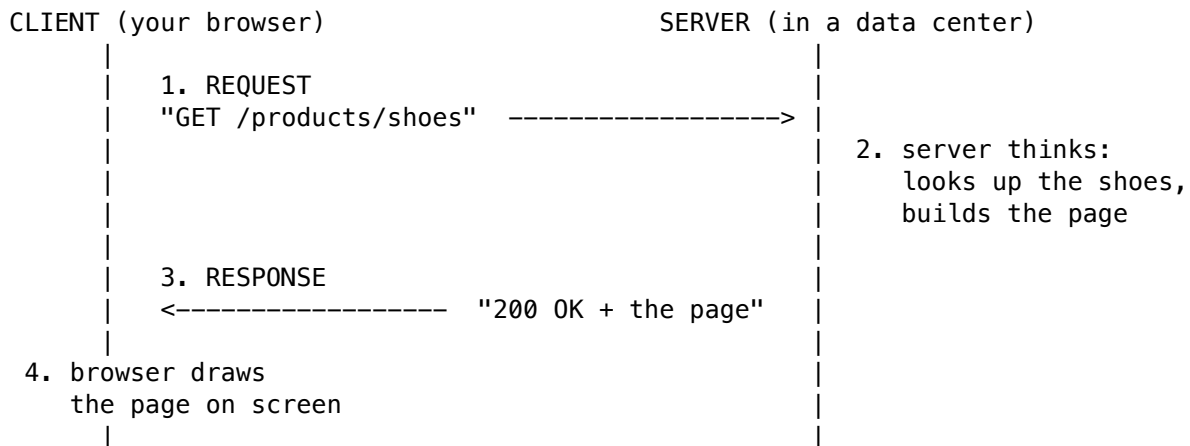
Léela como una dirección postal, al revés:

- El **scheme** (`https`), o esquema, es el protocolo: las reglas de la conversación. Hoy en día casi siempre es `https`.
- El **host** (`shop.example.com`) nombra a *qué* servidor contactar.
- El **path** (`/products/shoes`), o ruta, nombra *qué cosa* en ese servidor quieres.
- El **query** (`?size=10`) opcional, o consulta, lleva parámetros extra, como filtros o términos de búsqueda.

Cuando alguien dice “el endpoint” o “la ruta”, casi siempre se refiere a una ruta específica en un host específico. Eso es todo lo que significan esas palabras que asustan.

El ciclo de solicitud y respuesta

Aquí está el baile, de principio a fin. Escribes una dirección, y detrás de escena esto ocurre en una fracción de segundo:



Cuatro pasos, siempre:

1. El cliente envía una **solicitud** (request) a una URL.
2. El servidor la **procesa**: tal vez busca algo, ejecuta cierta lógica, verifica quién eres.
3. El servidor devuelve una **respuesta** (response): un estado (¿funcionó?) más algún contenido (la página, una imagen, algunos datos).
4. El cliente hace algo con la respuesta, normalmente dibujarla en tu pantalla.

Cargar una sola página web no es una sola solicitud: son docenas. El navegador pide la página, luego ve que necesita imágenes, tipografías y estilos, y dispara una solicitud nueva por cada uno. El mismo baile, muchas veces, muy rápido.

Métodos HTTP: los verbos

Cada solicitud lleva un **método**: un verbo que dice qué tipo de acción quieres. Hay varios, pero cuatro cubren casi todo, y encajan perfectamente con ideas cotidianas:

- **GET** — “dame esto”. Leer, obtener, ver. Cargar una página o una lista. GET nunca debería cambiar nada; solo recupera.
- **POST** — “aquí hay algo nuevo”. Crear. Enviar un formulario, registrarse, publicar un comentario.
- **PUT** — “actualiza esto”. Cambiar algo que ya existe, como editar tu perfil. (También oirás **PATCH** para actualizaciones parciales: la misma idea, con alcance más pequeño.)
- **DELETE** — “elimina esto”. Exactamente lo que suena: borrar un registro o un archivo.

Si no recuerdas nada más: **GET lee, POST crea, PUT actualiza, DELETE elimina**. Cuando tu app se comporta mal, saber qué verbo se envió suele ser la primera pista.

Códigos de estado: ¿funcionó?

Cada respuesta vuelve con un **código de estado** de tres dígitos: el veredicto de un vistazo del servidor sobre cómo salió la solicitud. Se agrupan por su primer dígito:

- **2xx — éxito**. Funcionó. 200 OK es el “aquí está lo que pediste” del día a día.
- **3xx — redirección**. “No está aquí, ve a mirar allá”. Común e inofensivo; tu navegador lo sigue automáticamente.
- **4xx — tú (el cliente) cometiste un error**. El famoso es **404 Not Found** (“no encontrado”): pediste una ruta que no existe. (También 401/403: no has iniciado sesión, o no tienes permiso.)
- **5xx — el servidor cometió un error**. **500 Internal Server Error** significa que el propio código del servidor falló o se rompió mientras manejaba tu solicitud.

Vale la pena tatuarte en el cerebro la división 4xx/5xx, porque te dice *dónde mirar*:

- Un **404** significa que la solicitud estaba mal, normalmente una URL incorrecta o una ruta con un error de tipeo. Mira qué se está pidiendo.
- Un **500** significa que el *servidor* se cayó. Tu solicitud estaba bien; el código del backend se rompió. Mira los registros (logs) del servidor, no la URL.

Cuando algo se rompe en una app hecha con vibe coding, “¿es un 4xx o un 5xx?” es una de las preguntas más rápidas para hacer un triaje.

status	code	
2xx	→ SUCCESS	it worked (200 OK)
3xx	→ REDIRECT	"go look over there" (auto-followed)
4xx	→ CLIENT error	YOUR request was wrong → check the URL/path
5xx	→ SERVER error	the backend broke → check server logs

Encabezados y cookies

Junto con el contenido principal, cada solicitud y respuesta lleva **headers** (encabezados): pequeñas notas etiquetadas con información extra. Son como lo escrito en un sobre, separado de la carta que contiene. Los encabezados dicen cosas como en qué formato está el contenido (Content-Type: text/html), o quién está preguntando.

Una **cookie** es un uso específico e importante de los encabezados. La web tiene memoria corta: por defecto, cada solicitud es independiente y el servidor te olvida en el instante en que responde. Una cookie arregla eso. Después de iniciar sesión, el servidor le entrega a tu navegador un pequeño token —una cookie— y tu navegador la envía automáticamente de vuelta con cada solicitud futura. Así es como un sitio “recuerda” que has iniciado sesión a través de varias páginas. Sin cookie, no hay memoria; tendrías que iniciar sesión en cada clic.

HTTPS y el candado

Has visto el pequeño candado en la barra de direcciones. Significa que la conexión usa **HTTPS** (HyperText Transfer Protocol Secure, “protocolo seguro de transferencia de hipertexto”): HTTP (HyperText Transfer Protocol, “protocolo de transferencia de hipertexto”) envuelto en una capa de cifrado llamada **TLS**.

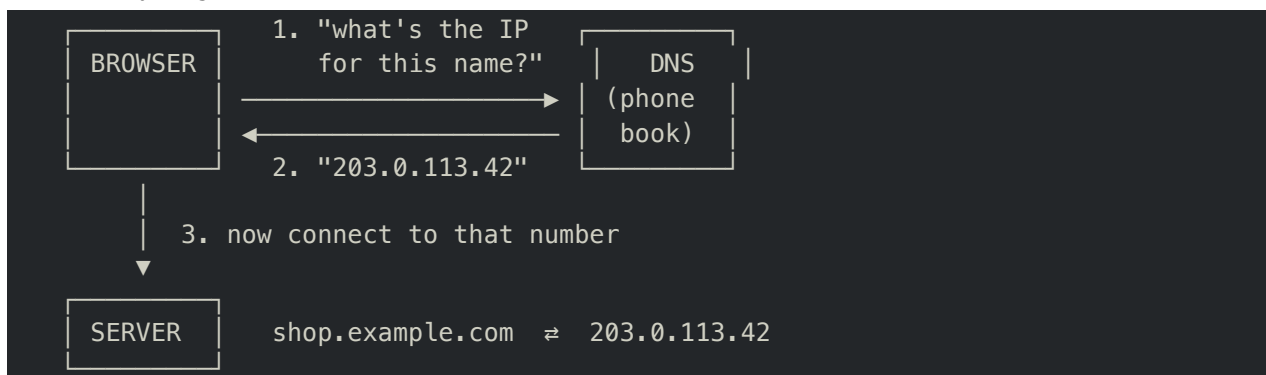
El HTTP simple envía todo como texto legible. Cualquiera que esté entre tú y el servidor —en una red Wi-Fi compartida, por ejemplo— podría leer tu contraseña mientras pasa, como una postal que cualquiera en la cadena de correo puede leer. HTTPS sella la conversación en un sobre que solo tú y el servidor pueden abrir. El candado confirma dos cosas:

- **Privacidad** — nadie en el medio puede leer lo que envías.
- **Identidad** — realmente estás hablando con el servidor que crees, no con un impostor.

La regla práctica es breve: las apps reales usan HTTPS, siempre, sin excepciones. Cualquier cosa que maneje inicios de sesión o pagos por HTTP simple está mal diseñada desde el origen.

De un nombre a un servidor: DNS

Un cabo suelto. Escribes `shop.example.com`, pero las computadoras en realidad no se encuentran entre sí por nombre: usan **direcciones IP** numéricas como `203.0.113.42`. **DNS** (Domain Name System, “sistema de nombres de dominio”) es la guía telefónica de la web: antes de que tu navegador pueda enviar una solicitud, le pregunta discretamente al DNS que traduzca el nombre de **dominio** amigable a la dirección numérica, y luego se conecta a esa.



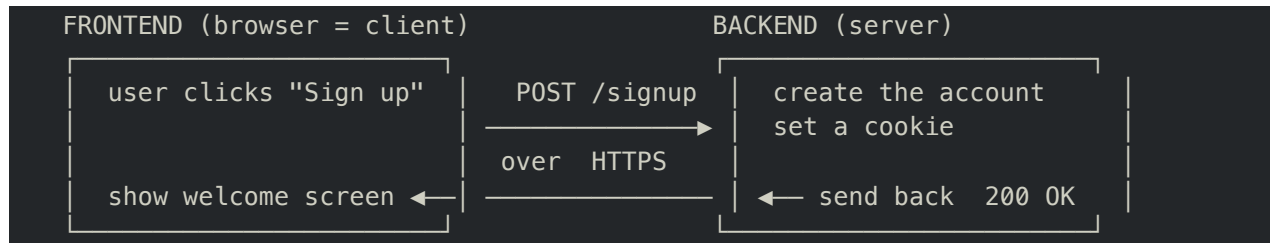
Compras un dominio, apuntas sus registros DNS al servidor donde corre tu app, y desde entonces el nombre lleva a la gente hasta tu máquina. Es un paso de búsqueda que casi nunca verás, pero es la razón por la que un nombre memorable puede representar a una cadena de números.

Cómo se relaciona esto con tu app

Ahora superpón todo este panorama sobre una app hecha con vibe coding, porque cada pieza que construirás vive en un lado del baile:

- El **frontend** es todo lo que corre en el navegador: los botones, formularios y el diseño que el usuario ve y sobre los que hace clic. Es el cliente. Sobre todo envía solicitudes y muestra respuestas.
- El **backend** es tu código corriendo en un servidor: la lógica que decide qué hacer con una solicitud, habla con la base de datos y construye la respuesta. Es el servidor.

Aquí está todo el recorrido de un registro en una sola imagen; cada palabra en negrita del capítulo está en el lugar que le corresponde:



Una interacción real cose todo esto junto. Un usuario hace clic en “Registrarse” (frontend). El navegador envía una solicitud **POST** por **HTTPS** a una ruta como `/api/signup`, llevando los datos del formulario. Tu backend la recibe, crea la cuenta, establece una **cookie** para que el usuario permanezca con la sesión iniciada, y devuelve un **200**. El frontend ve el éxito y muestra una pantalla de bienvenida. Cada palabra en negrita ahí es algo que acabas de aprender, y esa es la forma completa de casi todo lo que construirás.

No tienes que retener todo esto en la cabeza a la vez. Pero cuando un capítulo posterior diga “la solicitud llegó al servidor pero volvió con un 500”, o “configura este encabezado”, o “esta ruta maneja el POST”, ahora sabrás exactamente a qué parte del baile se refiere. Ese modelo mental es el verdadero prerrequisito, y ya lo tienes.

Repaso y práctica

Puntos clave

- La web es un baile sencillo repetido sin cesar: un **cliente** (normalmente un navegador) envía una **solicitud** (request), y un **servidor** devuelve una **respuesta** (response). Todo lo demás es detalle sobre esa base.
- Una **URL** son partes etiquetadas cosidas entre sí: esquema, host, ruta, consulta. El método es el verbo: **GET lee**, **POST crea**, **PUT actualiza**, **DELETE elimina**.
- Los **códigos de estado** son el veredicto de un vistazo del servidor. La división 4xx/5xx te dice dónde mirar: **4xx** significa que la solicitud estaba mal (URL incorrecta), **5xx** significa que el propio código del servidor se rompió.
- **HTTPS** (el candado) sella la conversación para que nadie en el medio pueda leerla: las apps reales lo usan siempre. Las **cookies** son cómo un sitio recuerda que tienes la sesión iniciada a través de las solicitudes.
- Tu **frontend** es el cliente (corre en el navegador); tu **backend** es el servidor (ejecuta tu lógica y habla con la base de datos). Cada función que construyas vive en un lado del baile.

Pruébalo

Abre cualquier sitio web, luego abre las herramientas de desarrollador de tu navegador (clic derecho ☐ “Inspeccionar”, luego la pestaña “Network” o “Red”) y recarga la página. Observa cómo se llena la lista de solicitudes: verás una solicitud de página seguida de docenas más para imágenes, tipografías y estilos, exactamente como se describió. Haz clic en una y mira su **método** (GET), su **código de estado** (esperemos que 200) y sus **encabezados**. Ahora estás viendo el baile de solicitud/respuesta ocurrir en tiempo real, con el vocabulario para leerlo.

Prompt del capítulo

I'm learning how the web works as a beginner. Using my own app as the example:
<describe your app, or paste a URL or an error you're seeing>

- Walk me through the request/response cycle for one specific action (like loading a page or submitting a form), step by step.
- Tell me which part is the client (frontend) and which is the server (backend).
- If I'm seeing a status code or error, explain whether it's a 4xx (my request was wrong) or a 5xx (the server broke), and where to look.
- Keep it concrete and beginner-friendly – no assumed knowledge.

APIs, JSON y datos



Casi ninguna aplicación que construyas vivirá sola. En el momento en que quieras mostrar un mapa, cobrar con una tarjeta, enviar un correo o consultar el clima, tu aplicación tiene que *hablar con la aplicación de otra persona*. El lenguaje que usa para eso se llama **API** (Application Programming Interface, interfaz de programación de aplicaciones), y las palabras que se envían de un lado a otro casi siempre están escritas en un formato llamado **JSON** (JavaScript Object Notation, notación de objetos de JavaScript). Este capítulo trata sobre ambos, no para que escribas la fontanería a mano, sino para que puedas leerla, revisar con sentido común lo que construyó la IA y notar cuándo algo está mal antes de que te cueste dinero.

No necesitas memorizar nada aquí. Necesitas reconocer las formas. Cuando la IA diga “voy a llamar a la API de Stripe y procesar la respuesta JSON”, deberías poder imaginar más o menos qué significa eso y

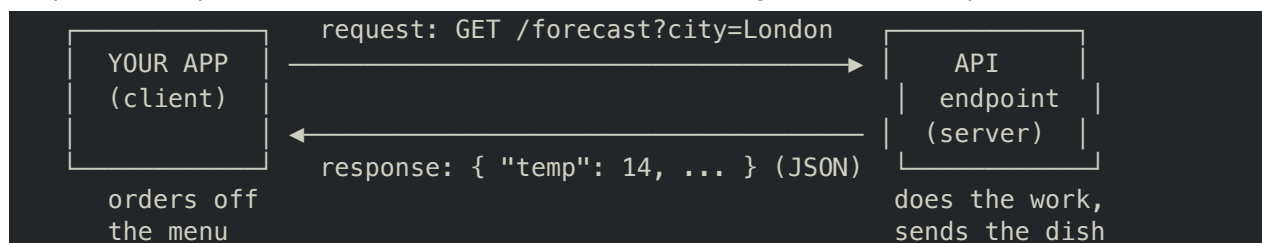
qué podría salir mal.

Qué es realmente una API

La forma más clara de pensar en una API es como un **menú de solicitudes que un programa le ofrece a otro**.

Un restaurante no te deja entrar a la cocina y cocinar. Te entrega un menú: aquí están las cosas que puedes pedir, así es como las pides, esto es lo que recibirás. No necesitas saber cómo funciona la cocina. Solo necesitas el menú. Una API es exactamente eso: una empresa dice “estas son las solicitudes que tienes permitido hacerle a nuestro servicio, y esto es lo que devuelve cada una”.

Cuando la IA escribe código que “llama a la API del clima”, está haciendo un pedido de ese menú: *dame el pronóstico para esta ciudad*. Los servidores de la empresa del clima hacen el trabajo y devuelven una respuesta. Tu aplicación nunca ve su base de datos ni su código. Solo recibe el plato.



El cambio de mentalidad importante: una API es un **contrato**. La otra empresa promete que si preguntas de una forma específica, obtendrás una respuesta con una forma específica. Tu trabajo —y el de la IA— es preguntar correctamente y leer la respuesta correctamente. La mayoría de los errores de API ocurren cuando un lado rompe ese contrato.

Endpoints y REST, en términos sencillos

Un **endpoint** (punto de acceso) es un solo elemento del menú. Es simplemente una URL (Uniform Resource Locator, localizador uniforme de recursos) que hace una cosa específica. Por ejemplo:

- `https://api.weather.com/forecast` — obtener un pronóstico
- `https://api.weather.com/cities` — listar las ciudades disponibles

Misma cocina, platos diferentes. Cada endpoint es una capacidad.

La mayoría de las APIs con las que trabajarás siguen un estilo relajado llamado **REST** (Representational State Transfer, transferencia de estado representacional). No necesitas la definición académica; necesitas dos ideas:

- **La URL nombra una cosa** (un “recurso”) — un usuario, un pedido, un pronóstico.
- **El verbo dice qué quieres hacer con ella**. Estos verbos son métodos HTTP (HyperText Transfer Protocol, protocolo de transferencia de hipertexto), los mismos que vienen del capítulo sobre cómo funciona la web:
 - GET — leer algo (obtener un pronóstico). Seguro; no cambia nada.
 - POST — crear algo (hacer un pedido, registrar un usuario).
 - PUT / PATCH — actualizar algo que ya existe.
 - DELETE — eliminar algo.

Entonces GET `/orders/42` significa “lee el pedido número 42”, y DELETE `/orders/42` significa “elimínalo”. Mismo sustantivo, verbo diferente, efecto completamente distinto. Cuando leas documentación de API o un diff, esas dos palabras —el **método** y la **URL**— te dicen la mayor parte de lo que hace una solicitud.

Qué es JSON

Cuando tu aplicación le hace una pregunta a una API, la respuesta vuelve como texto en un formato llamado **JSON**. Se convirtió en el predeterminado porque es legible por humanos y fácil de manejar para los programas. Lo verás constantemente, así que vale la pena poder leerlo.

JSON está construido a partir de **claves y valores**. Una clave es una etiqueta entre comillas; un valor es el dato asociado a ella. Aquí hay un ejemplo real que describe a un usuario:

```
{
  "id": 42,
  "name": "Ada Lovelace",
  "email": "ada@example.com",
  "isAdmin": false,
  "loginCount": 137,
  "tags": ["beta", "early-access"],
  "profile": {
    "city": "London",
    "avatarUrl": null
  }
}
```

Léelo como un formulario que alguien llenó. "name" es un campo, "Ada Lovelace" es lo que escribieron en él. Dos estructuras hacen todo el trabajo:

- Un **objeto** está envuelto en llaves { }. Es un conjunto de pares clave/valor, como un registro. Todo el ejemplo de arriba es un objeto.
- Un **arreglo** (array) está envuelto en corchetes []. Es una lista ordenada de valores. Arriba, "tags" contiene un arreglo de dos cadenas de texto.

Nota que "profile" es un objeto *dentro* del objeto. JSON anida: un valor puede ser a su vez un objeto o un arreglo, tan profundo como haga falta. Así es como una sola respuesta puede describir a un usuario, su dirección y sus últimos diez pedidos al mismo tiempo.

Dibujado como un árbol, el mismo objeto de usuario muestra su forma de un vistazo: campos planos arriba, una lista y un objeto anidado ramificándose:

```
{ } user (object)
├── id ..... 42 (number)
├── name ..... "Ada Lovelace" (string)
├── isAdmin ..... false (boolean)
├── tags ..... [ ] (array)
│   ├── "beta"
│   └── "early-access"
└── profile ..... { } (object)
    ├── city ..... "London" (string)
    └── avatarUrl . null (null = no value)
```

Los tipos de datos que encontrarás

Cada valor en JSON pertenece a un conjunto pequeño de tipos. Conocerlos te permite detectar cuándo un valor es del tipo equivocado, una fuente frecuente de errores.

- **string (cadena de texto)** — texto, siempre entre comillas dobles: "Ada Lovelace", "ada@example.com".
- **number (número)** — un número simple, sin comillas: 42, 3.14. (Las comillas cambian el significado: "42" es el *texto* cuatro-dos, no el número.)

- **boolean (booleano)** — true o false, sin comillas. Se usa para indicadores de sí/no como "isAdmin".
- **null** — un deliberado “aquí no hay valor”. "avatarUrl": null significa que el usuario no tiene avatar, algo distinto de que la clave falte por completo.
- **array (arreglo)** — una lista ordenada: ["beta", "early-access"].
- **object (objeto)** — un conjunto de pares clave/valor: el bloque "profile".

Ese es todo el vocabulario. El error más común en el mundo real es la trampa cadena-versus-número: el código espera el número 42 pero la API envía la cadena "42", y una comparación falla en silencio. Cuando algo se comporta de forma extraña, revisar el *tipo* de un valor suele ser la solución.

Una solicitud y respuesta reales

Así es como se ve una llamada real a una API de principio a fin: la solicitud que tu aplicación envía y el JSON que recibe de vuelta:

```
GET /forecast?city=London HTTP/1.1
Host: api.weather.com
Authorization: Bearer sk_live_8f4b2c...
Accept: application/json
```

```
{
  "city": "London",
  "unit": "celsius",
  "current": {
    "temp": 14,
    "condition": "cloudy"
  },
  "forecast": [
    { "day": "Mon", "high": 16, "low": 9 },
    { "day": "Tue", "high": 18, "low": 11 }
  ]
}
```

Esa línea Authorization es el siguiente tema, y el que más vale la pena hacer bien.

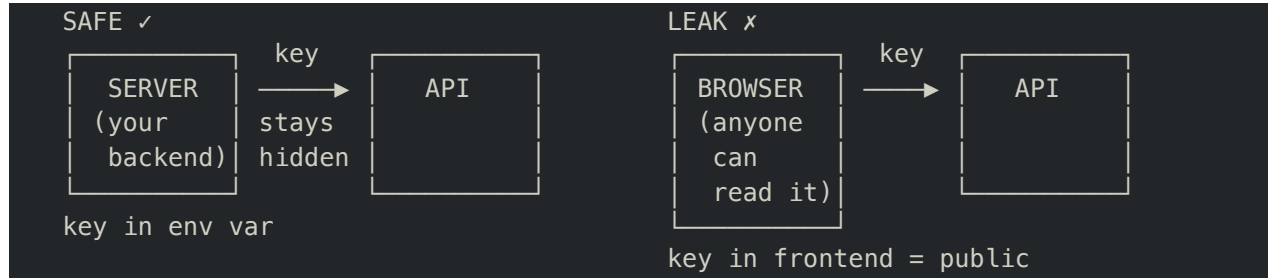
Claves de API y encabezados de autenticación

La mayoría de las APIs útiles no están abiertas al mundo. La empresa necesita saber *quién* está preguntando, para poder rastrear el uso y cobrarte. Te dan una **clave de API** (API key): una cadena secreta larga que identifica tu cuenta. La adjuntas a cada solicitud, normalmente en un **encabezado HTTP** (HTTP header) llamado Authorization, como la línea Bearer sk_live_... de arriba.

Trata una clave de API exactamente como una contraseña, porque eso es lo que es. Cualquiera que la tenga puede hacer solicitudes *como si fueras tú* y disparar tu factura o leer tus datos. Esto se conecta directamente con el capítulo de seguridad, y hay una regla que importa más que ninguna otra:

- **Nunca pongas una clave de API en código que se envía al navegador.** Todo lo que está en tu frontend es público: los usuarios pueden abrir las herramientas de desarrollo y leerlo. La IA, si le pides que “llame a la API desde la página”, pegará alegremente tu clave secreta justo ahí. Detenla. Las claves secretas pertenecen al *servidor*.
- **Nunca subas una clave a git.** Aunque la borres después, queda para siempre en el historial, y los bots escanean repositorios públicos buscando claves en cuestión de minutos.
- **Guarda las claves en variables de entorno**, no en el código mismo; el mismo patrón que cubre el capítulo de seguridad.

La clave debe viajar desde un lugar seguro, nunca desde el navegador. Imagina los dos caminos: uno seguro, uno una fuga:



Si alguna vez ves una clave real en un diff, en un archivo del frontend o en un mensaje de chat, trátala como ya filtrada: róta la (genera una nueva, desactiva la antigua) de inmediato.

Límites de tasa y la trampa de la factura sorpresa

Otras dos cosas que el menú no anuncia a gritos pero que sin duda te van a morder.

Límites de tasa (rate limits). Las APIs limitan cuántas solicitudes puedes hacer en una ventana de tiempo, digamos 100 por minuto. Si te pasas, te rechazan con un error 429 Too Many Requests hasta que bajes el ritmo. Esto existe para evitar que un solo cliente sature el servicio. La trampa: el código escrito por IA suele llamar a una API dentro de un bucle sin ninguna pausa, sobrepasando el límite al instante. Si una función funciona con un solo elemento pero falla al ejecutarla con mil, un límite de tasa es el principal sospechoso.

APIs medidas y de pago. Muchas APIs cobran por solicitud: fracciones de centavo cada una. Eso no es nada para unas pocas llamadas, y una factura real cuando algo se descontrola en un bucle. La historia de terror clásica: un error reintenta para siempre una llamada de pago fallida, o un formulario público permite que desconocidos disparen llamadas de pago, y te despiertas con una factura de cuatro cifras. Protégela:

- Lee los precios *antes* de conectar una API de pago. Conoce el costo por llamada.
- Establece un **límite de gasto o alerta de presupuesto** en el panel del proveedor, si lo ofrecen. La mayoría lo hace.
- Sospecha de cualquier código que llame a una API de pago dentro de un bucle o en cada carga de página.

Cómo leer la documentación de una API

Vivirás en la documentación de APIs, así que aquí tienes cómo obtener lo que necesitas rápido sin leerla toda. Para cualquier endpoint, busca cinco cosas:

- **El método y la URL** — GET /forecast. Qué hace, en dos tokens.
- **Autenticación** — qué encabezado o clave necesita. Normalmente hay una sección compartida para toda la API.
- **Parámetros** — las entradas que puedes enviar (aquí, city), cuáles son obligatorias y sus tipos.
- **Un ejemplo de respuesta** — la forma del JSON que obtendrás de vuelta. Esto es oro; cópialo.
- **Errores** — qué vuelve cuando algo sale mal, y los códigos de estado (401 no autorizado, 404 no encontrado, 429 límite de tasa alcanzado).

La buena documentación incluye un ejemplo de solicitud y respuesta que puedes copiar y pegar. Esos dos fragmentos te dicen más que párrafos de prosa: tómalos y dáselos a la IA.

Dejar que la IA haga la llamada, mientras tú verificas la forma

Aquí está la división práctica del trabajo. La IA es genuinamente buena escribiendo código que llama a APIs: conoce las librerías, la sintaxis de los encabezados, el manejo de errores. Déjala hacerlo. No necesitas escribir esta fontanería a mano.

Pero tú eres dueño de algo en lo que la IA es mala: **saber si la forma de los datos es realmente correcta**.

Un buen flujo de trabajo:

- **Dale a la IA la documentación real.** Pega el ejemplo de solicitud y respuesta de la documentación en tu prompt. Que la IA adivine la forma de una API de memoria es una de las principales fuentes de errores: inventará claves que parecen plausibles pero no existen. Los ejemplos reales la anclan a la realidad.
- **Pídele que primero te muestre la respuesta cruda.** “Antes de procesarla, registra el JSON real que devuelve la API.” Luego compárala a simple vista con lo que el código espera. ¿La clave que lee (`response.current.temp`) realmente existe en la respuesta? La mitad de los errores de API son un nombre de clave que no coincide.
- **Revisa los tipos con sentido común.** ¿Eso es un número o una cadena de texto? ¿Un campo que debería estar siempre presente a veces es `null`? Ya sabes leer JSON, úsalo.
- **Maneja el camino infeliz.** Pregunta: “¿Qué pasa si esta llamada falla, se agota el tiempo de espera, o alcanza el límite de tasa?” Si la respuesta es “la aplicación se cae”, eso es un error, no una característica.

No necesitas escribir la solicitud. Necesitas leer la respuesta y preguntar “¿es esta la forma que esperaba?” Ese único hábito —comparar los datos con el contrato— atrapa la mayoría de los errores de integración antes de que tus usuarios los vean jamás.

Resumen y práctica

Puntos clave

- Una **API** es un menú de solicitudes que un programa le ofrece a otro, y es un **contrato**: pregunta de la forma acordada, obtén una respuesta con la forma acordada. La mayoría de los errores de API ocurren cuando un lado rompe ese contrato.
- **JSON** es cómo vuelve la respuesta: claves y valores, contruidos a partir de objetos `{ }` y arreglos `[ ]`, que se anidan. Los tipos de datos son cadena de texto, número, booleano, `null`, arreglo, objeto.
- La **trampa cadena-versus-número** (42 frente a “42”) y la diferencia entre falta y `null` son los errores de forma más comunes. Ya sabes leer JSON: revisa el tipo cuando algo se comporte de forma extraña.
- Una **clave de API** es una contraseña. Nunca la envíes al navegador, nunca la subas a git, guárdala en variables de entorno. Una clave filtrada debe rotarse de inmediato.
- Presta atención a los **límites de tasa** (429) y a la **facturación medida**: los bucles escritos por IA que llaman a una API de pago sin pausa son la forma en que terminas con una factura de cuatro cifras. Lee los precios y establece una alerta de presupuesto primero.

Practicalo

Busca una API gratuita, sin necesidad de clave, en tu navegador; por ejemplo, abre `https://api.github.com/users/octocat` directamente en una pestaña nueva. Verás una respuesta JSON cruda. Léela como un formulario: identifica los objetos `{ }`, encuentra un arreglo si lo hay, y nombra el tipo de tres valores distintos (¿cuáles son cadenas de texto? ¿números? ¿booleanos? ¿hay algo que sea `null`?). Acabas de hacer exactamente la revisión de sentido común que este capítulo te pide aplicar a cada respuesta de API que conecte la IA.

El prompt del capítulo

```
I'm calling an API and I want to check the data shape myself, as a beginner.
Here are the real docs (example request + example response):
<paste the example request and JSON response from the API's docs>
```

- Before writing parsing code, show me the raw JSON the call returns and point out the keys, their types, and anything that might be null or missing.
- Confirm the keys my code reads actually exist in that response.
- Make sure the API key is read from an environment variable on the server, never hardcoded or sent to the browser.
- Tell me what happens if the call fails, times out, or hits a rate limit.

Anatomía de una aplicación



Cuando le pides a una IA que “agregue una página de inicio de sesión” o “guarde esto en la base de datos”, en silencio está tomando decisiones sobre media docena de piezas móviles. Si no sabes que esas piezas existen, las respuestas suenan como un idioma extranjero — y no puedes distinguir una buena de una confundida.

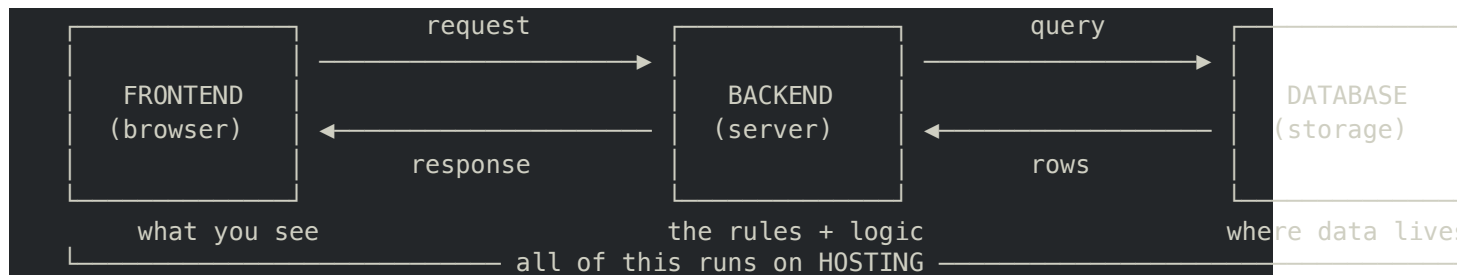
Este capítulo es un mapa. No te convertirá en ingeniero, pero te permitirá señalar cualquier pieza de una aplicación moderna y decir qué hace y dónde vive. Una vez que tengas el mapa, cada capítulo siguiente deja de ser abstracto: sabrás *en qué* casilla vive un problema antes de empezar a describirlo. Lo mantendremos simple — sin jerga sin definición, y un solo ejemplo recurrente (un usuario haciendo clic en un botón) que rastrearemos por cada capa.

El panorama general

Casi toda aplicación que construyas está hecha de cuatro partes que se comunican entre sí:

- **Frontend** — lo que el usuario ve y toca. Las pantallas, botones y textos dentro de un navegador o una app de teléfono.
- **Backend** — la lógica que se ejecuta en un servidor que tú controlas. Aplica las reglas, hace el trabajo y decide quién puede hacer qué.
- **Base de datos** — donde vive la información para que siga ahí mañana. Cuentas, publicaciones, pedidos.
- **Hosting (alojamiento)** — las computadoras, en algún lugar del mundo, que realmente ejecutan todo lo anterior para que otras personas puedan acceder a ello.

Así es como se conectan:



El frontend nunca habla directamente con la base de datos. Siempre pasa por el backend, donde viven las reglas. Ese único hecho explica mucho sobre cómo se construyen las aplicaciones — y por qué “simplemente deja que la página lea la base de datos” casi nunca es la respuesta correcta.

Frontend, backend, base de datos, hosting

Vamos una capa más profundo, usando nuestro ejemplo: un usuario hace clic en **“Guardar perfil.”**

Frontend. La aplicación visible, que se ejecuta en un navegador web o como app móvil. Dibuja el formulario, detecta el clic y recopila lo que el usuario escribió. Por diseño es *no confiable*: cualquiera puede abrirlo, inspeccionarlo y modificarlo. Así que su trabajo es verse bien y sentirse responsivo, no aplicar nada importante.

Backend. Cuando se hace clic en el botón, el frontend envía los datos al backend — código que se ejecuta en un servidor. El backend verifica que el usuario haya iniciado sesión, valida los datos (“¿esto realmente es una dirección de correo?”), aplica las reglas de negocio y solo entonces los guarda. Aquí es donde vive la confianza, porque los usuarios no pueden manipularlo.

Base de datos. El backend entrega los datos a la base de datos para almacenarlos. No es una caja mágica — piénsalo como un conjunto de hojas de cálculo muy estrictas (llamadas *tablas*) que se pueden buscar al instante y que nunca olvidan. Mañana, el backend vuelve a leer el perfil para mostrarlo de nuevo.

Hosting. En producción, nada de esto se ejecuta en tu laptop. Se ejecuta en computadoras alquiladas a un proveedor de hosting como Cloudflare, Vercel o una plataforma en la nube. El hosting es lo que hace que tu aplicación sea accesible en una dirección web real, en lugar de estar disponible solo en tu propia máquina.

Un instinto útil: cuando algo se rompe, pregunta *en qué capa*. “El botón no hace nada” es frontend. “Guarda lo incorrecto” suele ser backend. “Olvidó mis datos” apunta a la base de datos. Nombrar la capa es la mitad de arreglar el error — y la mitad de escribir un prompt sobre el que la IA pueda actuar.

HTML, CSS y JavaScript

El frontend, en un navegador, está construido a partir de exactamente tres lenguajes. No necesitas escribirlos a mano, pero sí necesitas saber qué *le corresponde* a cada uno.

HTML (HyperText Markup Language, Lenguaje de Marcado de Hipertexto) es la estructura — el contenido y su esqueleto. Dice “aquí hay un encabezado, aquí hay un párrafo, aquí hay un botón.” Son los sustantivos de la página; por sí solo, el HTML es un documento plano, sin estilo, como un archivo de texto con etiquetas. El navegador convierte tu HTML en un árbol vivo de objetos en memoria llamado el **DOM** (Document Object Model, Modelo de Objetos del Documento); cuando el código “cambia la página”, en realidad está cambiando el DOM, y el navegador vuelve a dibujar para reflejarlo.

CSS (Cascading Style Sheets, Hojas de Estilo en Cascada) es el estilo — cómo se ve todo. Colores, fuentes, espaciado, disposición, izquierda o centrado, pantalla pequeña versus grande. El CSS no cambia *qué* hay en la página, solo cómo se presenta. Si el HTML son los sustantivos, el CSS son los adjetivos.

JavaScript (JS) es el comportamiento — lo que sucede cuando interactúas. Escucha el clic, envía los datos al backend, muestra un indicador de carga, actualiza el DOM cuando llega la respuesta. HTML y CSS son estáticos; **JavaScript** hace que una página *haga* cosas. Son los verbos. (El mismo lenguaje también suele ejecutar el backend, por eso lo verás en todas partes.)

Una forma de imaginar la división: el HTML construye un árbol (el DOM), el CSS lo pinta, y JavaScript entra y lo cambia:



Paquetes y dependencias

Nadie escribe una aplicación completa desde cero. Los problemas difíciles y comunes — dar formato a fechas, gestionar pagos, renderizar un calendario — ya han sido resueltos y empaquetados por otras personas. Un **paquete** es un fragmento de código reutilizable que incorporas a tu proyecto en lugar de escribirlo tú mismo. Las **dependencias** de tu proyecto son la lista de paquetes de los que depende.

En el mundo de JavaScript, la herramienta que gestiona esto es **npm** (Node Package Manager, Gestor de Paquetes de Node). Dos archivos llevan la contabilidad:

- **package.json** — la lista legible para humanos de lo que tu proyecto necesita, más los comandos para ejecutarlo. Tú (o la IA) editan esto.
- **el archivo de bloqueo** (package-lock.json o similar) — un registro exacto, generado por máquina, de cada paquete y su versión precisa, para que la aplicación se construya de manera idéntica en tu máquina, en la de un compañero de equipo y en el servidor. No lo editas a mano; dejas que la herramienta lo gestione.

```
{
  "name": "my-app",
  "scripts": {
    "dev": "next dev",
    "build": "next build"
  },
}
```

```

"dependencies": {
  "next": "16.0.0",
  "react": "19.0.0"
}
}

```

¿Por qué importa? Cuando la IA dice “agreguemos una librería para esto”, está editando `package.json` y tú estás confiando en el código de otra persona — pocas dependencias bien conocidas son más seguras que un montón de dependencias oscuras. Y cuando una compilación se rompe misteriosamente, una dependencia desajustada es un sospechoso habitual, donde “eliminar y reinstalar los paquetes” es una solución real y común.

Variables de entorno y configuración

Algunos valores no se pueden escribir dentro de tu código: claves secretas, contraseñas de base de datos, la dirección de un servicio. Y algunos necesitan *cambiar* dependiendo de dónde se ejecute la aplicación. El código que pruebas en tu laptop (el entorno de **desarrollo**, o **dev**) debería hablar con una base de datos de prueba; la aplicación en vivo que tocan tus usuarios (el entorno de **producción**, o **prod**) debe hablar con la real — usando el *mismo* código.

La respuesta son las **variables de entorno** (a menudo “env vars”): valores nombrados que viven *fuera* del código y se le proporcionan cuando se ejecuta. El código dice “dame `DATABASE_URL`”, y el entorno decide cuál entregar.

```

# A config file (e.g. .env) – values live here, not in the code

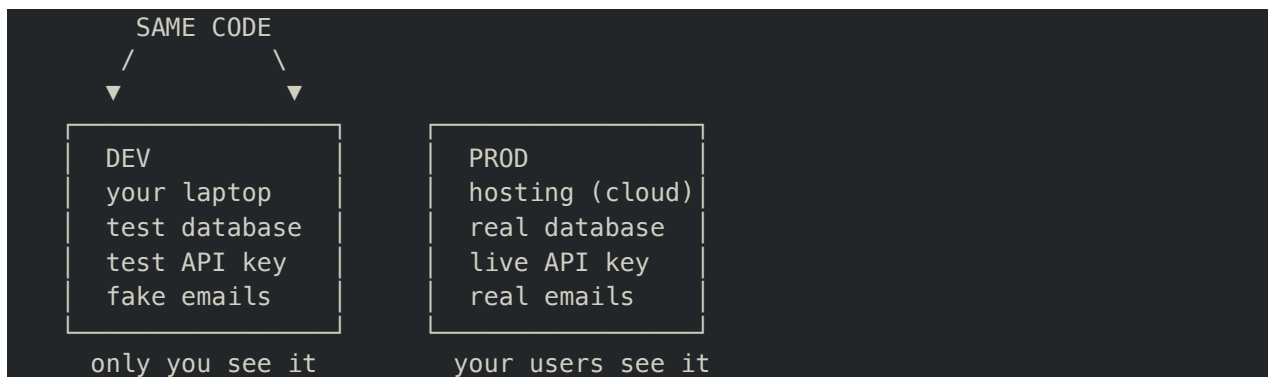
DATABASE_URL=postgres://localhost/myapp_dev
STRIPE_SECRET_KEY=sk_test_51H...      # dev = a test key
SEND_REAL_EMAILS=false                 # don't email real users while testing

```

Dos reglas te ahorrarán un verdadero dolor de cabeza:

- **Los secretos nunca van en el código.** Cualquier cosa en tu código puede terminar siendo pública (especialmente en un repositorio compartido), y una clave de pago o contraseña de base de datos filtrada es una emergencia real. Los secretos pertenecen a las variables de entorno, fuera del control de versiones.
- **Dev y prod son diferentes.** El punto es que el *mismo* código se comporte correctamente en ambos, porque el entorno — no el código — decide qué base de datos y qué claves usar.

Mismo código, dos hogares — las variables de entorno son lo que difiere:



La forma de un proyecto

Abre un proyecto típico y la lista de carpetas puede sentirse como ruido. No lo es — la mayoría de los proyectos siguen un patrón reconocible. Aquí está una aplicación web simplificada:

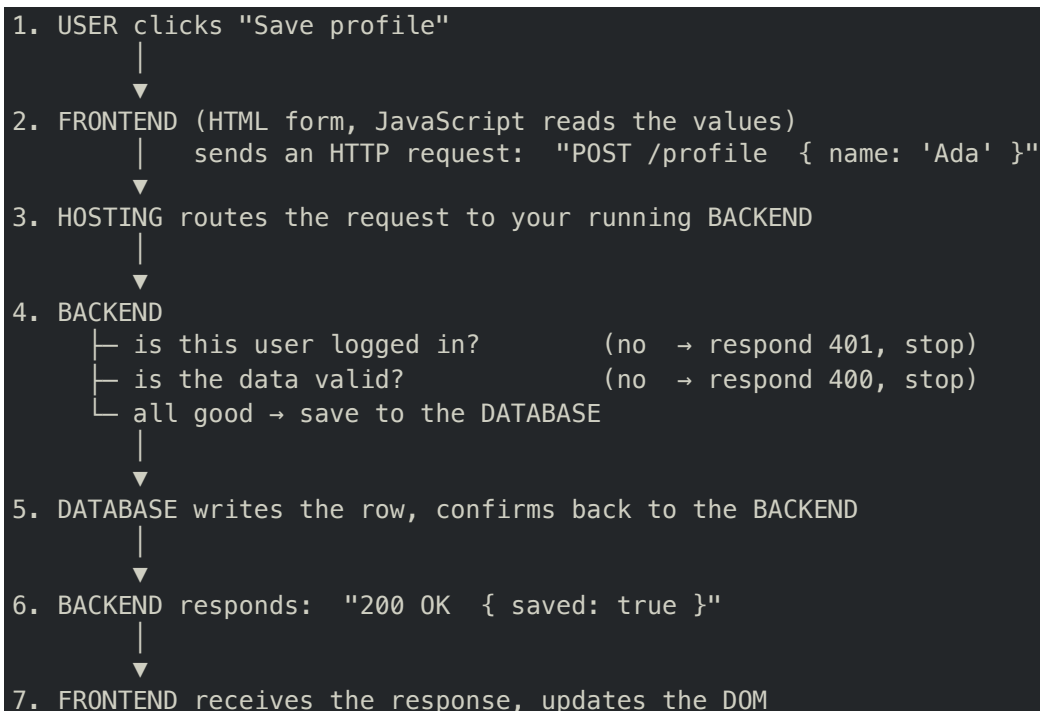
```
my-app/
├─ package.json          # dependencies + run commands (above)
├─ package-lock.json     # the lockfile – exact versions
├─ .env                  # local secrets/config – NOT shared
├─ .gitignore            # files Git should ignore (like .env)
├─ README.md             # what this project is, how to run it
├─ public/               # static files served as-is (images, icons)
├─ src/                  # your actual code lives here
│   ├─ components/       # reusable frontend pieces (a button, a card)
│   ├─ pages/ (or app/)  # the screens/routes users navigate to
│   ├─ lib/              # shared helpers and backend logic
│   └─ styles/           # CSS
```

No necesitas memorizar esto — solo reconocerlo, para que cuando la IA diga “voy a agregar esto a src/lib”, sepas más o menos dónde está eso y por qué. Algunas convenciones fundamentales:

- **src/** contiene el código que escribes. El desorden en el nivel superior es mayormente configuración.
- **Los archivos de configuración en la raíz** (el `package.json`, los archivos ocultos) configuran las herramientas, no el comportamiento de tu aplicación.
- **.gitignore** enumera archivos que *no* deben guardarse en el control de versiones — tu `.env` pertenece aquí, que es exactamente cómo los secretos se mantienen fuera de un repositorio compartido.

Cómo fluye una solicitud

Ahora lo juntamos todo. Sigamos a nuestro usuario haciendo clic en “**Guardar perfil**” a través de cada pieza que acabas de conocer. Este es el modelo mental más útil de todo el libro; si interiorizas un solo diagrama, que sea este.




```
| ("Profile saved ✓") – the user sees it
▼
DONE
```

Repasa lo que hizo cada capa: el **frontend** recopiló la entrada y la envió; el **hosting** llevó la solicitud al lugar correcto; el **backend** aplicó las reglas (inicio de sesión, validación) y decidió; la **base de datos** lo hizo permanente; y la respuesta fluyó de regreso hasta que el **frontend** pudo actualizar la pantalla. Toda aplicación es una variación de este ciclo, ejecutado miles de veces por segundo.

Nota que todas las verificaciones ocurren en el backend, nunca en el frontend — la regla de la primera sección, en acción: el frontend es para el usuario, el backend es para la verdad. Cuando sabes *dónde* ocurre cada paso, puedes describir una funcionalidad con precisión (“valida el correo en el backend antes de guardar”) en lugar de vagamente (“haz que el formulario funcione”) — y la precisión es todo el juego.

Recapitulación y práctica

Puntos clave

- Toda aplicación tiene cuatro partes: un **frontend** (lo que ven los usuarios), un **backend** (las reglas y la lógica), una **base de datos** (donde persisten los datos) y **hosting** (donde se ejecuta todo). El frontend nunca toca la base de datos directamente — pasa por el backend.
- En el navegador, **HTML** es la estructura, **CSS** es el estilo y **JavaScript** es el comportamiento; el código cambia la página cambiando el **DOM**.
- No escribes todo tú mismo — los **paquetes** (gestionados por **npm** a través de `package.json` y un archivo de bloqueo) proveen problemas ya resueltos como **dependencias**.
- Las **variables de entorno** mantienen los secretos fuera del código y permiten que el *mismo* código se comporte correctamente en dev y en prod.
- Una solicitud fluye de frontend → hosting → backend → base de datos y de regreso; nombrar la capa en la que vive un problema es la mitad de resolverlo.

Práctico

Elige una aplicación que uses a diario y narra en voz alta una acción a través de las capas: “Toco *enviar* (frontend) → va a su servidor (backend) → que verifica que estoy autorizado y guarda el mensaje (base de datos) → y el chat se actualiza (frontend).” Luego adivina en qué capa vive cada problema: el botón se ve desalineado; la aplicación dice “no has iniciado sesión”; un mensaje de ayer desapareció. Dominar este nombrado es la habilidad sobre la que se construye el resto del libro.

Prompt del capítulo

```
I'm learning how apps fit together. Here is a feature I want:
<one sentence describing what the user should be able to do>
```

```
Before writing any code, break it down by layer for me:
```

- Frontend: what the user sees and what gets sent
- Backend: what rules/checks must run, and where
- Database: what data needs to be stored or read
- Any env vars or config (especially secrets) involved

```
Then describe the request flow from click to result in numbered steps.
Keep it in plain language – I want to understand the shape before the code.
```

La mentalidad del vibe coding



Vibe coding es un término cargado de significados. Para algunos significa escribir un prompt de una línea, aceptar lo que sea que el modelo escupa y publicarlo sin leer ni una sola línea. Esa versión es un juguete. Produce demostraciones que se desmoronan en cuanto un usuario real las toca.

Este libro trata sobre la otra versión: dirigir a una IA para construir software de producción de la misma forma en que un ingeniero senior dirige a un equipo. Tú te mantienes a cargo de *qué* se construye y *por qué* es correcto. Delegas la escritura. Ese es el cambio, y es más grande de lo que suena.

Qué es realmente el vibe coding

Vibe coding es **construir software expresando la intención en lenguaje natural y dejando que una IA genere la implementación**, mientras tú diriges, revisas y refinas en un ciclo ajustado.

No es:

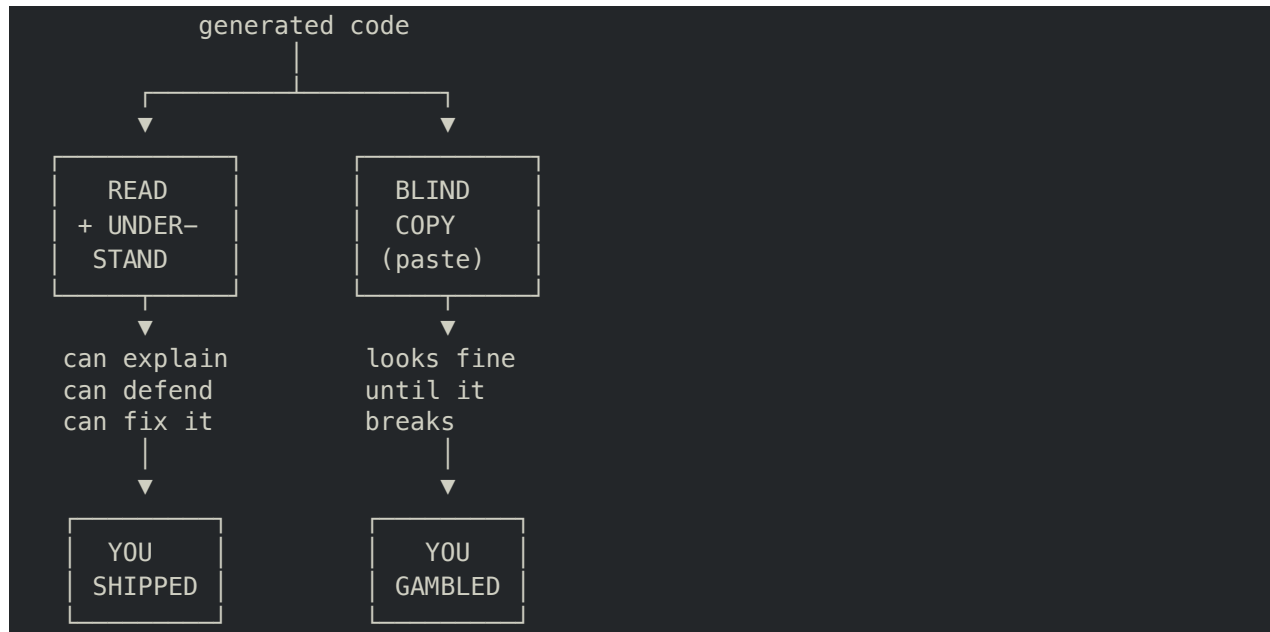
- Aceptar ciegamente código generado que no entiendes.
- Reemplazar el juicio de ingeniería con vibras.
- Un botón mágico que convierte una frase en un producto publicado.

Sí es:

- Tratar al modelo como un ingeniero junior rápido, conocedor y un poco demasiado confiado.
- Dedicar tu atención a la arquitectura, la corrección y el buen criterio en lugar de a la sintaxis.
- Iterar en minutos sobre cosas que antes tomaban horas.

El modelo mental que mejor funciona: tú eres el tech lead (el líder técnico), la IA es tu equipo. Un buen tech lead no escribe cada línea. Decide la dirección, establece restricciones, revisa el trabajo que su equipo somete para aprobación (en software, este lote propuesto de cambios se llama pull request o “solicitud de incorporación de cambios”) y detecta las malas ideas antes de que se publiquen. Tú haces lo mismo, solo que a la velocidad de una conversación.

Estos dos caminos parten del mismo código generado y terminan en lugares muy distintos:



Una palabra en esa definición hace todo el trabajo pesado: **entender**. Todo el enfoque descansa sobre una sola regla: *entiende lo que publicas*. No tienes que escribirlo tú, pero sí debes ser capaz de explicarlo, defenderlo en una revisión y arreglarlo a las 2 de la madrugada cuando te avisen que algo falló. Si no puedes, no hiciste vibe coding; apostaste y tuviste suerte. La diferencia entre esas dos cosas es el tema completo de este libro.

El cambio de mentalidad

Si venías de escribir cada carácter tú mismo, la parte más difícil no es aprender las herramientas. Es soltar el teclado.

Tu antiguo cuello de botella era la *producción*: qué tan rápido podías escribir código correcto. Tu nuevo cuello de botella es la *especificación* y la *revisión*: qué tan claramente puedes describir lo que quieres y qué tan bien puedes juzgar lo que regresa.

Esto cambia en qué te enfocas:

- **Antes:** memorizar APIs (Application Programming Interfaces, o interfaces de programación de aplicaciones: los menús de comandos que expone una herramienta), pelear con la sintaxis, escribir rápido.
- **Ahora:** describir la intención con precisión, descomponer problemas, evaluar el resultado de forma crítica.

Los ingenieros que prosperan con esto no son los que más rápido escriben. Son los que tienen opiniones firmes sobre cómo se ve “bueno” y la disciplina para rechazar resultados que no alcanzan ese nivel.

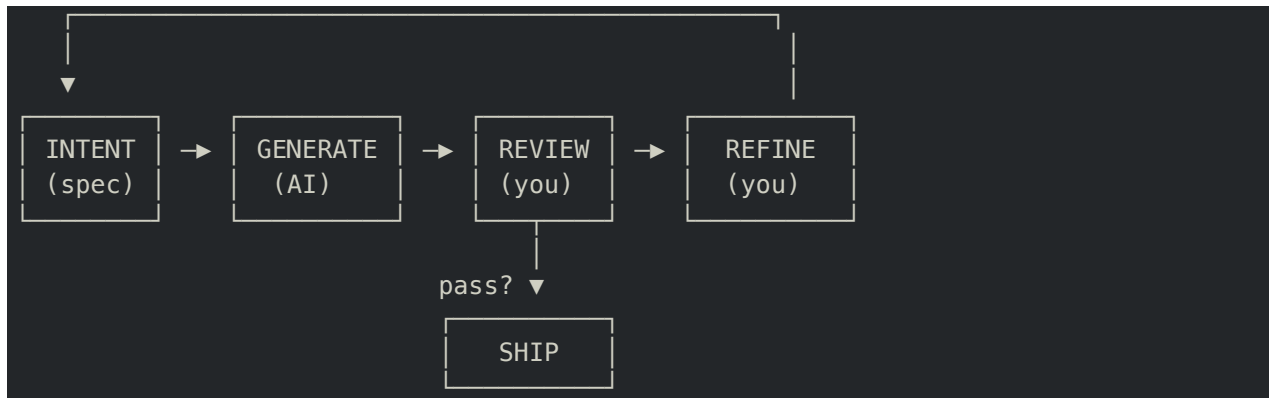
Hay un segundo cambio, más silencioso, que sorprende a la gente: tu relación con *no saber*. El instinto antiguo era detenerte y aprender la API antes de tocarla. Ahora, muchas veces puedes generar un primer borrador en una pila tecnológica que no conoces y aprender leyéndolo después. Eso es un superpoder real, y también una trampa. Leer código que tú no habrías podido escribir está bien. Publicar código que no

puedes *evaluar* no lo está. La línea divisoria es si puedes decir, con confianza, que el resultado es correcto. Apóyate en el modelo para escribir más rápido de lo que tú podrías; nunca te apoyes en él para saber más de lo que estás dispuesto a verificar.

El ciclo: intención □ generar □ revisar □ refinar

Todo en este libro regresa a un solo ciclo. Interiorízalo.

El ciclo gira hasta que el resultado alcanza tu estándar, y la revisión es la puerta por la que tiene que pasar cada vez:



1. **Intención.** Indica lo que quieres y las restricciones que importan: entradas, salidas, casos límite, la pila tecnológica, el estilo. Una intención vaga produce código vago.
2. **Generar.** Deja que el modelo escriba la implementación. No le guíes de la mano con la sintaxis; describe el comportamiento.
3. **Revisar.** Lee lo que regresó. ¿Realmente hace lo que debe? ¿Maneja la lista vacía, el valor nulo, la llamada de red fallida? ¿Es la versión *más simple* que funciona?
4. **Refinar.** Señala qué está mal, da una corrección concreta y vuelve a generar. Repite hasta que alcance tu estándar.

La mayoría de los principiantes se saltan el paso 3. Generan, corre, siguen adelante. Luego se rompe en producción con un caso que nunca describieron. El paso de revisión es donde ocurre la ingeniería.

Así se ve un buen primer prompt en este ciclo:

```
Write a TypeScript function `parseDuration(input: string): number`  
that converts strings like "1h30m", "45s", "2d" into total seconds.  
  
Requirements:  
- Support units: d (days), h (hours), m (minutes), s (seconds).  
- Multiple units can combine: "1h30m" = 5400.  
- Reject invalid input (empty, unknown units, negative) by throwing  
  an Error with a clear message.  
- No external libraries. Include 5 unit tests covering the edge cases.  
  
Show the function and tests only.
```

Fíjate en la estructura: una firma precisa, reglas explícitas, los casos límite señalados, las restricciones indicadas y una petición de pruebas. No estás esperando que el modelo adivine bien; estás eliminando la ambigüedad que produce malas suposiciones.

El paso de refinar es tan una habilidad como el primer prompt. Una retroalimentación vaga produce arreglos vagos. Compara estas dos correcciones:

```
Bad: "this is wrong, fix it"
```

```
Good: "parseDuration('1h30m') returns 90 instead of 5400 – you're
      summing the numbers but ignoring the unit multipliers. Multiply
      each value by its unit's seconds (d=86400, h=3600, m=60, s=1)
      before summing. Also add a test for the combined case '2d3h'."
```

La buena versión nombra el síntoma, la causa y la solución. Estás revisando como lo harías con el pull request de un colega, y entre más precisa sea tu señalización, menos rondas necesitarás. Cada ciclo debe *converger*. Si te encuentras dando vueltas en círculo, esa es una señal de que tu intención estaba subespecificada; deja de regenerar y reescribe la especificación en su lugar.

Dónde brilla

El vibe coding es genuinamente transformador para:

- **Andamiaje desde cero (greenfield).** Levantar una nueva app, conectar rutas, configurar el entorno: trabajo tedioso pero muy transitado.
- **Código repetitivo y de conexión (boilerplate y glue).** Endpoints CRUD, validación de formularios, transformaciones de datos, adaptadores entre dos APIs.
- **Terreno desconocido.** Un lenguaje o framework que no dominas. El modelo conoce los modismos; tú aportas el criterio.
- **Refactorizaciones y migraciones.** “Convierte estas promesas a async/await”, “divide este archivo de 400 líneas por responsabilidad”.
- **Exploración desechable.** Prototipos que reescribirás de todos modos. La velocidad importa más que el pulido.

En estas zonas, un prompt claro más una revisión cuidadosa le gana a escribir todo a mano, siempre.

El hilo común es que estas tareas son **muy transitadas y verificables**. El modelo ha visto diez mil endpoints CRUD, así que produce unos idiomáticos; y tú puedes saber de un vistazo si el resultado es correcto. Cuando ambas cosas son ciertas (patrón común, verificación barata), el vibe coding está cerca de ser ganancia pura.

Dónde falla

Sé honesto sobre los límites, porque aquí es donde el vibe coding sin supervisión le quema a la gente:

- **Lógica profunda y novedosa.** Algoritmos sutiles, concurrencia, cualquier cosa donde estar 95 % en lo correcto significa estar roto. El modelo está seguro de sí mismo incluso cuando se equivoca.
- **Cambios extendidos entre muchos archivos.** Pierde el hilo en una base de código grande y rompe silenciosamente cosas que no puede ver.
- **Seguridad y dinero.** Autenticación, pagos, permisos, cualquier cosa donde un error que se ve plausible tiene consecuencias reales. Revisa esto como si tu trabajo dependiera de ello, porque podría ser así.
- **Problemas subespecificados.** Si tú no sabes cómo se ve “correcto”, la IA no puede leerle la mente. Especificación basura entra, código basura sale.

El modo de falla es casi siempre el mismo: código que *parece* correcto, corre en el camino feliz y está mal de una forma que habrías detectado si lo hubieras leído. La solución es el paso de revisión. No hay forma de saltárselo.

Dos preguntas ubican cualquier tarea en este mapa: ¿el patrón es común? y ¿el resultado es barato de verificar?

EASY TO VERIFY

HARD TO VERIFY

COMMON PATTERN	PURE UPSIDE (scaffold, boilerplate)	go slower, demand tests
RARE / NOVEL	read it, but doable	DANGER ZONE (novel logic, auth, money)

Fíjate en el inverso de la regla anterior: estos son los casos que son *raros, sutiles o costosos de verificar*. El patrón es poco común, así que el modelo está adivinando; o el bug es invisible en el camino feliz, así que una revisión rápida no lo detectará. Eso no significa que evites la IA aquí; significa que vayas más despacio, reduzcas los pasos, exijas pruebas y leas cada línea como si fuera hostil.

Trampas comunes

Unas pocas trampas atrapan a casi todos en su primer mes:

- **Aceptar lo primero que corre.** “Funciona” y “es correcto” son afirmaciones distintas. La primera significa que el camino feliz no se cayó; la segunda significa que revisaste los bordes. No las confundas.
- **Mega-prompts.** Pedir una funcionalidad completa de una sola vez le da al modelo demasiado espacio para desviarse y te da demasiado que revisar de golpe. Descompón. Publica el fragmento revisable más pequeño posible y luego construye sobre él.
- **Dejar que el contexto se pudra.** A medida que una sesión crece, el modelo empieza a contradecir decisiones anteriores y a reintroducir bugs que ya habías arreglado. Cuando eso empieza, resume el estado actual y comienza de nuevo en lugar de pelear con un hilo confundido.
- **Confiar en la confianza como señal.** El tono del modelo es idéntico ya sea que tenga razón o esté alucinando. Su certeza no te dice nada. Tu verificación te lo dice todo.
- **Externalizar la decisión, no solo la escritura.** Está bien preguntar “¿cuáles son mis opciones para X?”. Es un error dejar que la respuesta elija tu arquitectura por ti sin que tu criterio esté en el ciclo.

El buen criterio es la nueva habilidad central

Cuando cualquiera puede generar código, la habilidad escasa es saber qué código generado es *bueno*.

El buen criterio es lo que te dice:

- Esta abstracción es prematura; pide la versión más simple.
- Esto maneja la demostración pero no la entrada real.
- Esto es técnicamente correcto pero una pesadilla de mantenimiento.
- Esto está *casi* bien, y aquí está lo específico que hay que cambiar.

Construyes buen criterio de la forma tradicional: leyendo mucho código, publicando cosas y viendo qué se rompe. La IA no reemplaza esa experiencia; la vuelve más valiosa, porque ahora tu juicio es el cuello de botella y el diferenciador. La persona con buen criterio publica software sólido rápido. La persona sin él publica rápido y luego se ahoga en bugs.

Aquí está el corolario incómodo: la IA eleva el piso para todos, lo que significa que *aplana* la ventaja de conocer la sintaxis y *afila* la ventaja de tener buen juicio. Los junior que se apoyan en ella para saltarse el aprendizaje se estancarán, porque nunca desarrollan el criterio que les dice cuándo el resultado está mal. Los que la usan para publicar más y leer más se irán acumulando ventaja. Misma herramienta, resultados opuestos, y la diferencia está enteramente en si te mantienes involucrado con lo que regresa.

Expectativas realistas

Establece esto ahora para no decepcionarte después:

- **Vas a leer mucho código.** Revisar es el trabajo. Si no quieres leer código, esto no es un atajo para evitarlo.
- **No va a resolver nada no trivial de un solo intento.** Espera unas cuantas rondas del ciclo. Eso es normal, no un fracaso.
- **Sigues siendo responsable.** Cuando código escrito por IA falla en producción, “el modelo lo escribió” no es una defensa. Tú lo publicaste.
- **Se vuelve más rápido, no gratis.** Las herramientas son extraordinarias. No son magia. El juicio sigue siendo tuyo.

Y para ser directo sobre lo que este libro *no* es: no es un camino para publicar software que no entiendes, y no es un esquema para hacerte rico rápido. Cualquiera que venda el vibe coding como “construye una app sin aprender a programar” está vendiendo la versión de juguete, la que se desmorona al contacto con usuarios reales. La habilidad se acumula para las personas que la tratan como una palanca sobre el juicio de ingeniería, no como un reemplazo de este.

La promesa del vibe coding es real: puedes construir más, más rápido, en más dominios de los que podrías tú solo. Pero recompensa al ingeniero que se mantiene involucrado, que dirige, revisa y refina, y castiga al que se desconecta y acepta lo que sea que aparezca.

Sé el tech lead, no el espectador. El resto de este libro trata sobre cómo.

Repaso y práctica

Puntos clave

- El vibe coding es dirigir a una IA para construir software mientras tú te mantienes a cargo de *qué* y *por qué*, no publicar ciegamente código que no puedes explicar.
- La regla que sostiene todo es *entiende lo que publicas*: no tienes que escribirlo tú, pero debes poder defenderlo y arreglarlo.
- Tu cuello de botella se movió de la producción (escribir rápido) a la especificación y la revisión (describir con claridad, juzgar con criterio).
- Vive en el ciclo: intención → generar → revisar → refinar, y nunca te saltes el paso de revisión, donde ocurre la ingeniería real.
- La confianza del modelo no es una señal; tu verificación sí lo es. El buen criterio y el juicio son las habilidades escasas que se acumulan.

Práctico

Elige una función pequeña que entiendas genuinamente, algo como “validar que una contraseña cumple una política” o “formatear una cantidad de bytes como una cadena legible para humanos”. Escribe un prompt preciso para ella (firma, reglas, casos límite, una petición de pruebas), génerala, y luego ejecuta deliberadamente el paso de *revisión*: rastrea al menos un caso límite a mano y confirma que el código lo maneja. Si no lo hace, escribe un prompt de refinamiento preciso (síntoma + causa + solución) en lugar de “arréglalo”. El objetivo es sentir el ciclo completo una vez, de principio a fin, sobre código que puedas evaluar por completo.

Prompt del capítulo

```
Act as my pair-programming partner. I'll describe a small, self-contained function. Before you write any code, restate the requirements and list the edge cases you'll handle so I can confirm them. Then write the function plus unit tests covering those edge cases.
```

```
Function: <one-line description>
```

Inputs / outputs: <types and shapes>

Constraints: <stack, libraries to use or avoid, performance limits>

After the code, point out any assumption you made that I should verify.
Keep it to the simplest version that satisfies the requirements.

Configura tu espacio de trabajo con IA



La programación vibe (vibe coding) es una colaboración. Tú aportas la intención y el criterio; la IA aporta velocidad y una capacidad de escritura infatigable. Pero, como toda colaboración, funciona mejor con un entorno compartido, reglas claras y un lugar seguro donde equivocarse. Este capítulo trata de construir ese entorno antes de escribir una sola función.

Un buen espacio de trabajo hace tres cosas: le da a la IA el contexto que necesita para hacer cambios inteligentes, te permite ver los resultados de esos cambios rápidamente, y hace que cada cambio sea fácil de deshacer. Si aciertas con esto, la IA se siente como una colaboradora senior. Si fallas, se siente como una máquina tragamonedas. La diferencia rara vez está en el modelo; está en la habitación en la que lo colocas. Un agente capaz en un repositorio caótico, sin pruebas, sin archivo de contexto y sin control de versiones, se descontrolará; uno modesto en un proyecto limpio y bien instrumentado te sorprenderá. La configuración es el multiplicador.

Nada de esto es trabajo que haces una vez y olvidas. Trata tu espacio de trabajo como algo que ajustas continuamente: cada vez que la IA te sorprenda de mala manera, pregúntate si faltó una pieza de contexto,

si el ciclo de retroalimentación fue lento, o si un valor por defecto inseguro lo permitió; luego arregla la habitación, no solo la línea de código.

Elige un editor o asistente nativo de IA

Tu primera decisión es dónde vive la IA. Hoy existen aproximadamente tres categorías, y no tienes que comprometerte para siempre.

- **Editores nativos de IA** (Cursor, Windsurf): un editor de código familiar con la IA profundamente integrada. Ideal si quieres ver y tocar el código mientras la IA trabaja a tu lado.
- **Agentes de terminal/CLI** (Claude Code, Codex CLI, Gemini CLI, Aider): piensa en escribir comandos en un prompt en lugar de hacer clic en botones; ese panel de control basado solo en texto es la línea de comandos, o CLI (Command-Line Interface, Interfaz de Línea de Comandos). La IA se ejecuta como un agente de línea de comandos que lee archivos, los edita y ejecuta comandos. Es lo mejor para dejar que la IA trabaje de forma más autónoma a través de muchos archivos.
- **Plugins de IDE** (GitHub Copilot, Continue, JetBrains AI): un IDE (Integrated Development Environment, Entorno de Desarrollo Integrado) es la aplicación todo en uno en la que la mayoría de los programadores escriben código; estos son asistentes complementarios dentro de un editor que ya te gusta.

Para lanzar software real, elige uno que pueda **leer todo tu proyecto, editar varios archivos y ejecutar comandos** (pruebas, compilaciones, linters) por su cuenta. La capacidad de ejecutar y verificar su propio trabajo es lo que separa a un asistente útil de un simple autocompletado. Una herramienta que solo puede sugerir la siguiente línea te deja haciendo toda la verificación a mano; una herramienta que puede ejecutar la prueba que acaba de modificar cierra el ciclo por sí misma.

Más allá de esa capacidad esencial, hay algunas cosas que vale la pena sopesar antes de decidirte:

- **Ventana de contexto y conciencia del proyecto** — ¿puede realmente cargar los archivos relevantes, o se olvida del comienzo de la conversación a mitad de una tarea?
- **Modelo de permisos** — ¿pregunta antes de ejecutar comandos y editar archivos, o actúa primero? Ambos tienen su lugar; entiende en qué modo estás.
- **Costo y límites** — los agentes autónomos pueden consumir uso rápidamente. Entiende los precios antes de dejar que uno funcione durante una hora.

No te obsesiones con la elección; las habilidades de este libro se transfieren entre herramientas. La mayoría de los programadores vibre experimentados terminan manteniendo dos instaladas: un agente de terminal rápido para trabajo masivo y un editor gráfico para revisión. Una GUI (Graphical User Interface, Interfaz Gráfica de Usuario) es el tipo de programa familiar con ventanas y botones que se maneja apuntando y haciendo clic; una TUI (Text User Interface, Interfaz de Usuario de Texto) es su prima solo de texto que vive en la terminal. La sección final del capítulo sobre TUI frente a GUI retoma exactamente ese emparejamiento.

Estructura tu proyecto para que la IA pueda navegarlo

Los modelos de IA razonan sobre tu código leyéndolo. Un proyecto desordenado, disperso e inconsistente es tan difícil para ellos como lo es para ti. No necesitas una arquitectura perfecta, pero unos pocos hábitos dan resultado de inmediato:

- **Mantén una estructura predecible.** El código fuente en un lugar, las pruebas junto al código fuente o reflejando su estructura, la configuración en la raíz.
- **Usa nombres claros y descriptivos.** `calculateMonthlyInvoiceTotal` es mejor que `calc2`. La IA usa los nombres como pistas.
- **Prefiere archivos más pequeños.** Un archivo de 200 líneas es más fácil de cambiar correctamente que uno de 2.000, tanto para ti como para la IA.
- **Agrupar lo relacionado.** Cuando el código, los estilos y las pruebas de una función viven juntos, la IA reúne el contexto en una sola lectura.

- **Mantén un punto de entrada claro.** Un único lugar evidente donde comienza la aplicación le da a la IA un hilo del que tirar cuando está mapeando el proyecto.

Una estructura típica y amigable para la IA podría verse así:

```
my-app/
├── AGENTS.md           # project rules & context for the AI
├── README.md          # what the project is, how to run it
├── .env.example        # documents needed secrets (no real values)
├── .gitignore          # excludes .env, build output, node_modules
├── package.json        # scripts: dev, test, lint, build
├── src/
│   ├── features/
│   │   └── invoices/   # code + tests for one feature, together
│   ├── lib/            # shared helpers
│   └── index.ts        # entry point
└── tests/              # cross-cutting / integration tests
```

El beneficio se acumula: cuanto más fácil sea navegar tu proyecto para la IA, más pequeños y quirúrgicos se vuelven sus cambios. Un repositorio bien organizado le permite a la IA tocar tres archivos relevantes en lugar de andar adivinando entre treinta, lo que se traduce en diferencias (diffs) más pequeñas, menos accidentes y revisiones que realmente puedes leer. Si alguna vez ves que la IA hace ediciones amplias y sin foco, la estructura suele ser la verdadera culpable, no el prompt.

Escribe un archivo de contexto (AGENTS.md)

Lo más rentable que puedes hacer es escribir un **archivo de contexto** que la IA lea automáticamente. Distintas herramientas buscan distintos nombres: AGENTS.md, CLAUDE.md, .cursorrules, o un .github/copilot-instructions.md, pero la idea es idéntica: un documento vivo que le dice a la IA cómo funciona *tu* proyecto y cómo quieres *tú* que se comporte.

Sin él, la IA adivina. Con él, la IA deja de reformatear tu código, deja de inventar bibliotecas que no usas y deja de volver a explicar decisiones que ya tomaste. Piénsalo como documentación de incorporación para una nueva compañera de equipo que es brillante pero tiene amnesia total entre sesiones: todo lo que necesita para ser productiva tiene que vivir en algún lugar que lea cada vez.

Un archivo inicial práctico:

```
# Project: Invoice Manager

## What this is
A small web app for freelancers to create and track invoices.

## Tech stack
- TypeScript + React (Vite)
- Tailwind for styling
- SQLite via Prisma
- Vitest for tests

## Commands
- Install: `npm install`
- Run dev server: `npm run dev`
- Run tests: `npm test`
- Lint: `npm run lint`
```

```
## Conventions
- Use functional components and hooks; no class components.
- Keep components small; one component per file.
- Write a test for every new function in `src/lib`.
- Never edit files in `src/generated/` — they are auto-generated.

## Don'ts
- Don't add new dependencies without flagging it first.
- Don't commit secrets; use environment variables.
```

Mantenlo breve y actualizado. Un archivo de contexto que crece hasta mil líneas deja de leerse con atención, ya sea por ti o por el modelo, así que pódalo con la misma agresividad con la que lo amplías. Cuando la IA repita el mismo supuesto erróneo, esa es tu señal para añadir una línea; cuando una regla ya no coincide con la realidad, elimínala antes de que empiece a confundir activamente al agente.

Unos pocos patrones separan un archivo de contexto que funciona de uno que se ignora:

- **Sé específico y concreto.** “Usa nuestras convenciones” es inútil. “Usa zod para validación; no usa-mos yup” es una regla que la IA realmente puede seguir.
- **Declara los “no hagas” con la misma fuerza que los “haz”.** Los modelos están ansiosos por ayudar, lo que significa que añadirán con gusto una dependencia o reescribirán una configuración a menos que les digas que no lo hagan.
- **Señala ejemplos.** “Sigue el estilo de src/features/invoices” le da a la IA un modelo concreto que copiar en lugar de una instrucción vaga que interpretar.

Trata el archivo de contexto como un registro de cada lección que de otro modo tendrías que repetir. Es la inversión más barata y duradera de todo este capítulo.

Convierte el control de versiones en un reflejo

El control de versiones es tu red de seguridad, y con la IA se vuelve esencial. La IA puede cambiar diez archivos en cinco segundos; necesitas una forma limpia de inspeccionar, conservar o descartar ese trabajo.

- **Confirma (commit) a menudo, en porciones pequeñas.** Antes de pedir un cambio grande, confirma lo que funciona. Un árbol de trabajo limpio significa que siempre puedes volver a él.
- **Lee el diff antes de confirmar.** Este es tu verdadero paso de revisión. No tienes que entender cada línea, pero deberías entender qué cambió y por qué.
- **Usa ramas para trabajo arriesgado.** Crea una rama, deja que la IA se desate y fusiona solo si te gusta el resultado. Si no, elimínala; no pasa nada.
- **Escribe mensajes de confirmación honestos.** Tanto tu yo futuro como la IA leen el historial para entender la intención.

El cambio mental: los commits no son burocracia, son puntos de guardado en un videojuego. Guarda antes de la pelea contra el jefe final. Y aprende las dos vías de escape que hacen que valga la pena mantener un historial limpio: `git checkout .` (o `git restore .`) para descartar experimentos sin confirmar, y `git revert` para deshacer un commit que ya hiciste. Saber que siempre puedes volver a un estado conocido y funcional es exactamente lo que te permite darle a la IA una correa larga sin ansiedad.

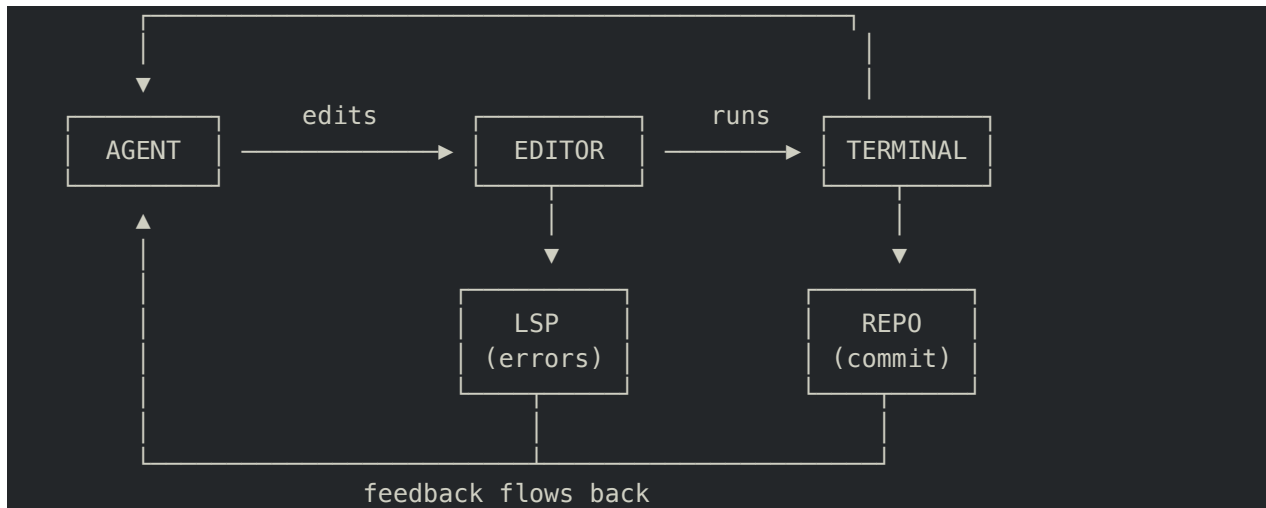
Mantén el ciclo de retroalimentación ajustado

Cuanto más rápido puedas ver si un cambio funciona, más rápido convergen tú y la IA en algo correcto. Un ciclo lento —donde esperas minutos para saber si algo se rompió— multiplica silenciosamente cada error.

- **Haz que “ejecutarlo” sea un solo comando.** `npm run dev`, `make test`, lo que sea, documéntalo en tu archivo de contexto para que la IA también pueda ejecutarlo.

- **Apóyate en pruebas rápidas.** Incluso un puñado de pruebas unitarias le da a la IA una forma de verificar su propio trabajo sin que la vigiles.
- **Usa un linter y un formateador.** Detectan toda una clase de errores al instante y mantienen las discusiones de estilo fuera de tus revisiones.
- **Muéstrale los errores a la IA.** Cuando algo se rompa, pega el mensaje de error real. La retroalimentación específica produce soluciones específicas; “no funciona” produce conjeturas.

Las piezas de un espacio de trabajo con IA forman un solo ciclo conectado: el agente edita el repositorio, el editor y el servidor de lenguaje reportan problemas, la terminal ejecuta el código, y cada resultado regresa al agente:



La versión más profunda de esta idea es dejar que la IA cierre el ciclo por sí misma. En lugar de que tú ejecutes el comando, leas el fallo y lo transmitas, dale al agente permiso para ejecutar pruebas y el verificador de tipos después de cada cambio y reaccionar a lo que ve. El ciclo se convierte en: editar, ejecutar, leer el fallo, corregir, ejecutar de nuevo, todo sin que tú estés en medio. Tu trabajo pasa de mensajero a director: fijas el objetivo y las barreras de seguridad, y luego observas girar el ciclo. Cuando la IA puede editar, ejecutar, ver el fallo y corregirlo por su cuenta, has construido un ciclo que hace trabajo real mientras tú permaneces en la silla de director.

Gestiona los secretos con cuidado

Las herramientas de IA leen tus archivos, y a veces comparten contexto con un servicio remoto. Eso convierte a las claves filtradas en un riesgo real. Construye buenos hábitos desde el primer día:

- **Nunca pongas claves reales en el código.** Usa variables de entorno y un archivo `.env`.
- **Siempre ignora tu `.env` en git.** Confirma un `.env.example` con marcadores de posición vacíos para que la IA sepa qué se necesita sin ver los valores.
- **Delimita las claves de forma estrecha** y rota cualquiera que pueda haberse expuesto.
- **Dile a la IA en tu archivo de contexto** que nunca codifique ni confirme secretos.

Una regla contundente como `Never read or print the contents of .env` en tu `AGENTS.md` es un seguro barato. También lo es un hook de pre-commit o una herramienta de escaneo de secretos que rechace un commit que contenga algo que parezca una clave; la defensa en profundidad importa aquí porque el costo de una filtración es asimétrico. Una clave subida a un repositorio público queda comprometida en el momento en que aterriza, incluso si la borras segundos después, porque los bots rastrean los commits en tiempo real. Ante la duda, asume que una clave expuesta está quemada y róta; rotar una clave toma un minuto, mientras que limpiar el desastre después de una filtración puede tomar días.

Una configuración local sensata hace seguros los cambios de la IA

Únelo todo con un entorno donde los errores sean baratos y recuperables:

- Trabaja en una **rama**, no directamente en `main`.
- Mantén el árbol de trabajo **limpio antes** de cambios grandes, para que el diff sea significativo.
- Ten las pruebas y un servidor de desarrollo **listos para ejecutar** con un solo comando.
- Mantén los secretos **fuera del repositorio** y el archivo de contexto **actualizado**.

Con esta base, el peor caso para cualquier cambio de la IA es `git checkout` y un nuevo comienzo. Esa seguridad es lo que te permite moverte rápido y dirigir con audacia, sabiendo que nada de lo que hace la IA es realmente irreversible. En el próximo capítulo, comenzaremos a usar este espacio de trabajo para construir de verdad.

Diagnósticos en vivo (LSP)

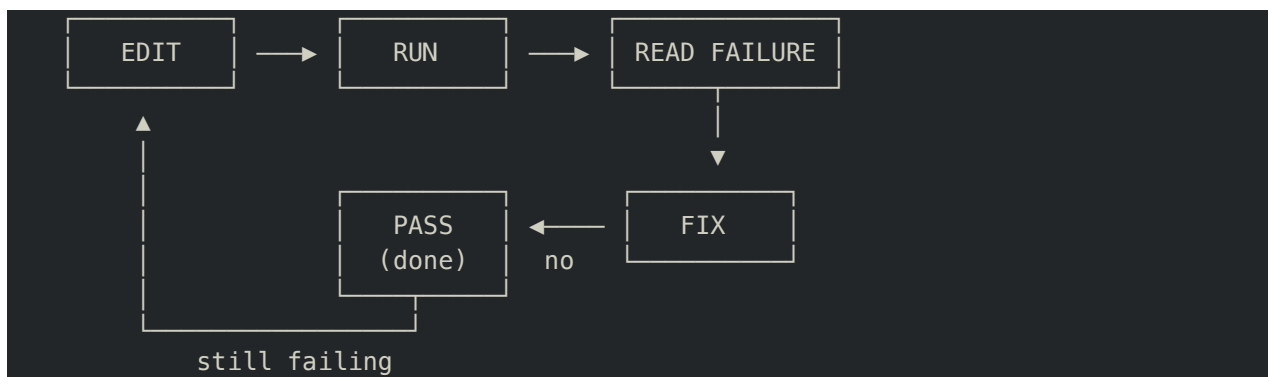
El ciclo de retroalimentación más ajustado no espera a que ejecutes el código; atrapa los errores en el momento en que se escriben. Eso es lo que provee el **Language Server Protocol (Protocolo de Servidor de Lenguaje, LSP)**: la “inteligencia” en vivo detrás de tu editor, que incluye **errores** en tiempo real, **información de tipos**, **ir a la definición** y **autocompletado**. Un servidor de lenguaje lee tu código a medida que cambia y reporta problemas antes de que nada se ejecute.

El punto de apalancamiento para la programación vibe es **retroalimentar esos diagnósticos a la IA**. Sin eso, el ciclo es lento: la IA escribe código, tú lo ejecutas, falla por un error de tipeo o de tipo, y pegas el mensaje de vuelta a mano. Con los diagnósticos conectados, el agente ve la misma subrayada roja que ves tú en el instante en que escribe la línea, y corrige el error de tipo o de lint *sobre la marcha*, antes de que el código llegue siquiera a ejecutarse.

Puedes darle al agente esta visión de varias formas:

- Ejecuta el servidor de lenguaje de tu stack (tsserver, rust-analyzer, pyright, etcétera); la mayoría de los editores nativos de IA hacen esto por ti.
- Haz que el agente ejecute un comando de verificación de tipos o de lint y lea la salida, o expón los diagnósticos a través de un servidor MCP.
- Trata un informe de diagnósticos limpio como parte del “terminado”, no como una ocurrencia tardía.

Conectado de esta manera, el agente ejecuta el mismo ciclo autocorrectivo por su cuenta: editar, ejecutar, leer el fallo, corregir y volver a ejecutar hasta que los diagnósticos vuelvan limpios:



Esta es también la razón por la que un lenguaje tipado puede ser un superpoder silencioso para la programación vibe: cada anotación de tipo es una pequeña especificación verificada por máquina que atrapa los errores de la IA antes de que lleguen a ejecutarse jamás. Un ciclo de ejecutar-fallar-pegar se convierte en uno silencioso y autocorrectivo. La IA da menos vueltas y converge más rápido.

Extendiendo tu agente

Recién salido de la caja, un asistente de IA es genérico. El mismo modelo base se convierte en una colaboradora mucho mejor una vez que envuelves herramientas a su alrededor, transformando un chatbot simple en un **arnés (harness)** ajustado a tu proyecto. Los principales puntos de extensión que hay que conocer:

- **Skills (habilidades)** — capacidades empaquetadas y reutilizables (instrucciones más scripts) que el agente carga a demanda, de modo que tu conocimiento de dominio está disponible sin tener que volver a explicarlo cada sesión.
- **Plugins** — paquetes que envían skills, comandos, hooks y conectores juntos como una sola unidad instalable.
- **MCP (Model Context Protocol, Protocolo de Contexto de Modelo)** — un estándar abierto para conectar el agente a herramientas y datos externos (bases de datos, APIs (Application Programming Interfaces, Interfaces de Programación de Aplicaciones — los menús estándar de comandos que otro software expone), tu sistema de archivos) a través de una interfaz común.
- **Subagentes** — agentes auxiliares especializados que el agente principal genera para trabajo puntual (búsqueda, revisión, una tarea secundaria larga), cada uno con su propia ventana de contexto.
- **Hooks (ganchos)** — scripts que se disparan automáticamente en eventos del ciclo de vida (antes de que se ejecute un comando, después de que se edite un archivo), para que puedas hacer cumplir reglas o inyectar contexto de forma determinista.
- **Comandos de barra** — flujos de trabajo cortos y nombrados que invocas con `/nombre`, colapsando una petición repetida de varios pasos en una sola pulsación de tecla.

No necesitas todo esto desde el primer día. Empieza simple y añade una skill o un comando de barra en el momento en que notes que te estás repitiendo la misma configuración. Una regla práctica: la tercera vez que escribas la misma instrucción de varios pasos, conviértela en un comando de barra; la primera vez que desees que el agente pueda acceder a una herramienta que no puede, busca un servidor MCP para ella. Los hooks, en particular, vuelven a aparecer en el capítulo de barreras de seguridad como una forma de *hacer cumplir* tus reglas en lugar de simplemente esperar que la IA las recuerde.

macOS frente a Windows

La mayoría de las herramientas de IA —agentes de CLI, scripts, tutoriales— asumen silenciosamente un shell Unix. Eso condiciona la elección práctica del entorno de desarrollo.

- **macOS** es **nativo de Unix**. Viene con un shell POSIX real y los mismos `git`, `ssh` y `curl` que usa el resto del mundo del desarrollo. La mayoría de los agentes y scripts “simplemente funcionan”.
- **Windows** es totalmente capaz, pero el camino más fluido es **WSL2** (Windows Subsystem for Linux, Subsistema de Windows para Linux): un entorno Linux real dentro de Windows. Haz tu programación con IA *dentro* de WSL2 en lugar de pelear con las peculiaridades nativas de PowerShell.
- **Linux** es el tercer miembro de la familia y se comporta casi exactamente igual que macOS para estos propósitos; es Unix, así que los mismos agentes y scripts se ejecutan sin cambios. Si ya estás en él, estás listo.

Algunos detalles a tener en cuenta para mantener tu trabajo portable en todos ellos:

- **Rutas** — `/home/tu/proyecto` frente a `C:\Users\tu\proyecto`; las barras invertidas y las letras de unidad rompen los scripts. WSL2 evita la mayor parte de esto.
- **Finales de línea** — Windows usa CRLF, Unix usa LF. Los desajustes causan diffs ruidosos y scripts de shell rotos; configura `core.autocrlf` y un `.gitattributes` para normalizar a LF.
- **Sensibilidad a mayúsculas y minúsculas** — macOS y Windows a menudo tratan `File.ts` y `file.ts` como lo mismo; Linux y los sistemas de integración continua (CI) no, así que el código que compila localmente puede fallar en la nube. Sé consistente con las mayúsculas en las importaciones y los nombres de archivo.
- **Permisos** — los modos de archivo de Unix (`chmod +x`) no se traducen limpiamente a Windows.

Sea cual sea tu sistema, pídele a la IA que mantenga los comandos portables —barras diagonales hacia adelante, finales de línea LF, mayúsculas consistentes— para que sus scripts se ejecuten igual en tu laptop y en CI. Cuando la IA te entregue una línea de comando que asume una herramienta que no tienes, la solución más rápida suele ser decirle tu sistema operativo y shell en el archivo de contexto para que deje de adivinar.

TUI frente a GUI

Las herramientas de programación con IA vienen en dos formas, y una programadora vibe productiva recurre a ambas. Los **agentes de terminal / TUI** (Claude Code, Codex CLI, Aider) se ejecutan en la terminal: rápidos, programables y componibles, ideales para automatización y para dejar que la IA trabaje de forma autónoma a través de muchos archivos. Las **herramientas de GUI / editor** (Cursor, Windsurf, extensiones de IDE) viven en un editor visual: fáciles de descubrir, con diffs en línea, aceptación con un clic, y una curva de entrada suave; más fáciles para revisar cambios.

	TUI / terminal	GUI / editor
Velocidad	Muy rápida, poca sobrecarga	Más lenta, más interfaz
Programable	Sí — canalizar, encadenar, automatizar	Limitado
Diffs visuales	Texto plano	Ricos, en línea
Facilidad de descubrimiento	Baja (hay que conocer los comandos)	Alta (se puede hacer clic)
Mejor para	Automatización, autonomía, usuarios avanzados	Revisión, incorporación, trabajo visual

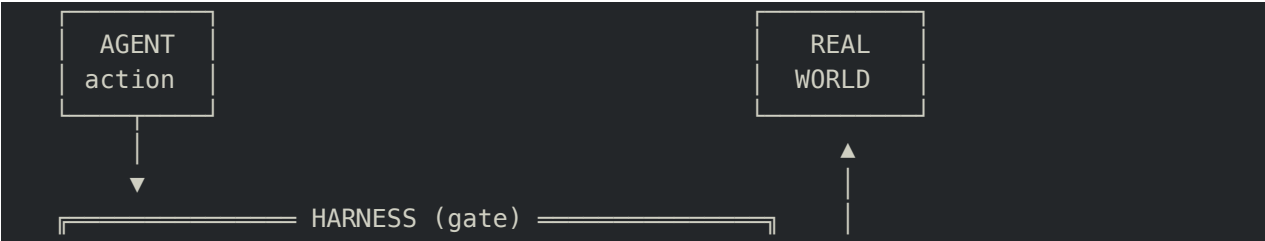
No tienes que elegir uno para siempre. La combinación es el punto clave: conduce el grueso del trabajo a través de un agente de TUI rápido, y luego entra a una GUI para revisión visual o ediciones de precisión. Un flujo de trabajo común es dejar que un agente de terminal haga los cambios amplios a través de muchos archivos, y luego abrir el diff lado a lado de la GUI para realmente leer y aprobar lo que hizo: velocidad para hacer, claridad para verificar. Elige una herramienta principal cómoda, y recuerda que las habilidades de este libro se transfieren entre ellas.

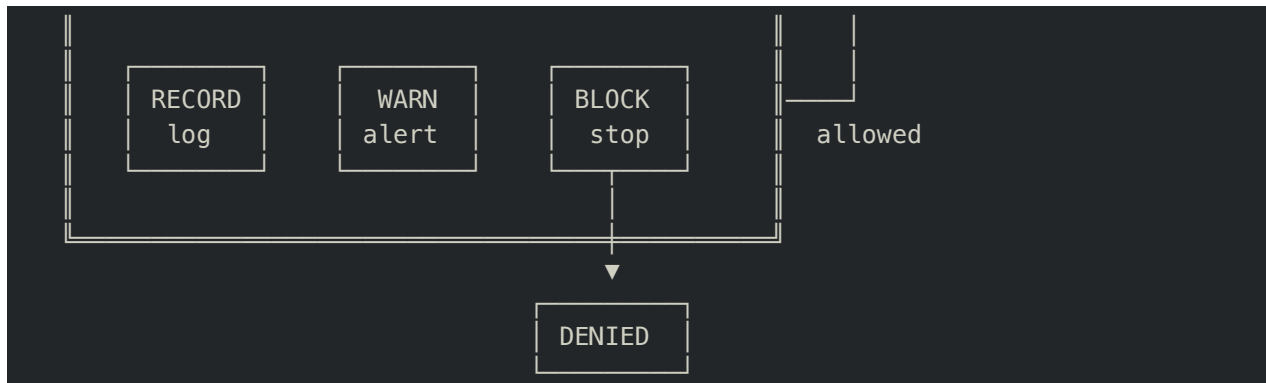
Ingeniería de arneses (Harness Engineering)

Los hábitos son algo que hay que recordar hacer. El siguiente peldaño al configurar tu espacio de trabajo es convertir esas verificaciones en algo que la IA **no pueda** saltarse. Eso es un arnés (harness).

Piensa en un arnés como un puesto de control de seguridad que se sitúa entre el agente y el mundo real. En cada paso peligroso hace una de tres cosas: registrar lo que pasó, advertirte, o bloquear la acción por completo. Antes de que se ejecute un comando, antes de que se escriba un archivo, cuando se envía un prompt, al confirmar (commit), al subir (push); hay una puerta. El agente no puede rodearla, como tú podrías olvidarte de ejecutar las pruebas a las 2 de la madrugada.

Cada acción peligrosa que realiza el agente pasa primero por el arnés, donde cada puerta puede registrarla, advertirte, o bloquearla antes de que llegue jamás al mundo real:





¿Por qué hacerlo obligatorio? Toda la apuesta de la programación vive es “deja que la IA corra, tú diriges”. Eso solo da frutos si la IA no puede irse por un precipicio mientras parpadeas. Un arnés desplaza tu seguridad de la *vigilancia* (leer cada diff, recordar cada regla) al *código* (reglas aplicadas automáticamente). Para cualquier cosa que pondrías frente a un usuario real, eso no es un adorno opcional; es lo que hace que la autonomía sea segura.

Un buen arnés sigue algunos principios que vale la pena conocer:

- **Silencioso en el éxito, ruidoso en el fallo** — no dice nada cuando todo está bien, así que una advertencia sigue significando algo.
- **Nunca autocorregir** — propone, bloquea o advierte; no “corrige” silenciosamente al agente y oculta el problema.
- **Retira reglas inactivas antes de añadir nuevas** — mantenlo ligero y confiable, no un montón creciente de verificaciones obsoletas.
- **Impulsado por configuración** — un solo motor, ajustes por proyecto, para que encaje en cualquier repositorio.
- **Registros de solo anexo (append-only)** — un registro a prueba de manipulaciones de lo que realmente hizo el agente.

No tienes que construir esto desde cero. Existen arneses listos para usar y de fácil integración; dancin-lab/harness es un motor agnóstico de proyecto que puedes conectar al ciclo de vida de herramientas de un agente de programación. Honestamente: un arnés no hace que la IA sea correcta. Hace que los errores de la IA sean baratos de detectar, lo cual, cuando te mueves rápido, es la mayor parte de la batalla.

Seguridad de costos: CLI frente a API

Una factura puede arruinar un proyecto, y rara vez proviene del hospedaje. Proviene de una API de IA medida por consumo. Las herramientas que configuras para programar día a día —Claude Code, Cursor y similares— suelen ser una **suscripción de tarifa plana**: el costo de un prompt más es efectivamente cero. En el momento en que tu *aplicación* llama a una **API o SDK** de IA sin procesar —siendo un SDK (Software Development Kit, Kit de Desarrollo de Software) la biblioteca de código preempaquetada que un proveedor te da para llamar a su servicio— la economía cambia por completo: se te cobra **por token, por solicitud**, sin techo. Un reintento con errores, un trabajo por lotes sin límite, o un agente que entra en bucle pueden convertir silenciosamente una factura de \$0 en reposo en una sorpresa de cuatro cifras de la noche a la mañana. Eso es el “susto de la factura” (“bill shock”), y es la forma más común en que los desarrolladores solitarios se queman.

Pon una jaula alrededor de cualquier API de pago antes de lanzarla:

- **Establece un tope de gasto duro** en el panel del proveedor; un techo real, no solo una alerta por correo.
- **Prefiere la CLI de tarifa plana para desarrollo**; guarda la API medida para la función de producción que realmente la necesita.

- **Limita la tasa de tus propias llamadas** — pon un tope a la concurrencia y a las solicitudes por minuto — y vigila el panel de uso los primeros días.
- **Usa modelos baratos/pequeños para trabajo masivo, y prueba primero con entradas diminutas** (un registro, no toda la tabla).
- **Nunca lances un bucle sin límite que llame a una API de pago.** Acota cada bucle.

Conviértelo en una regla desde el principio: *toda API de pago tiene un tope duro y un limitador de tasa.*

Resumen y práctica

Puntos clave

- Un buen espacio de trabajo le da a la IA contexto, un ciclo de retroalimentación rápido y una forma barata de deshacer; la configuración es el multiplicador, a menudo más que el modelo mismo.
- Un archivo de contexto (AGENTS.md / CLAUDE.md) es lo más rentable que puedes escribir: reglas concretas, prohibiciones contundentes, y referencias a ejemplos.
- Convierte el control de versiones en un reflejo — confirma en pequeño, lee el diff, usa ramas para trabajo arriesgado — para que nada de lo que haga la IA sea realmente irreversible.
- Mantén el ciclo ajustado y deja que el agente lo cierre: editar, ejecutar, leer el fallo, corregir. Retroalimenta a la IA con los diagnósticos (LSP, lint, verificación de tipos).
- Enjaula cada API de pago con un tope duro y un limitador de tasa, mantén los secretos fuera del repositorio, y usa un arnés para convertir las verificaciones de seguridad en algo que la IA no pueda saltarse.

Ponlo en práctica

Abre cualquier proyecto en el que estés trabajando (o crea uno vacío) y escribe a mano un primer borrador de AGENTS.md: qué es el proyecto, el stack tecnológico, los comandos exactos para ejecutar dev/test/lint, tres convenciones reales, y al menos dos prohibiciones contundentes (por ejemplo, “nunca leas ni imprimas .env”, “no añadas dependencias sin señalarlo primero”). Luego pídele a tu herramienta de IA que haga un cambio trivial y observa si respeta el archivo. Cada vez que haga algo que no querías, añade o afina una línea. Estás ajustando la habitación, no solo el código.

Prompt del capítulo

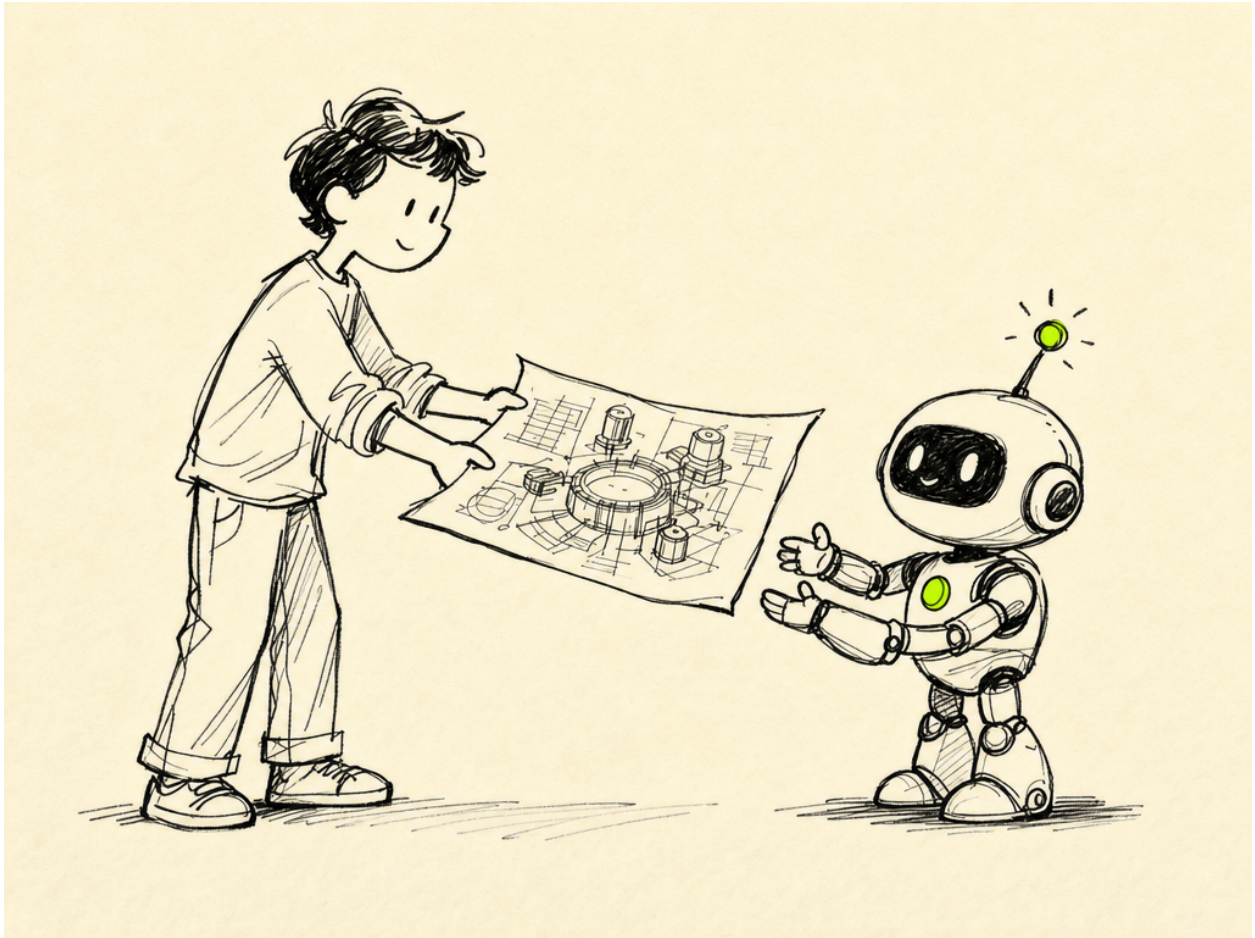
```
Read my project and help me write a concise AGENTS.md context file for it.
```

```
First, inspect the repo and tell me:
```

- the tech stack and versions you detect
- the commands for install / dev / test / lint / build
- the layout and where the entry point is

```
Then draft an AGENTS.md with these sections: What this is, Tech stack, Commands, Conventions, and Don'ts. Be specific and concrete (name the exact libraries and rules, not "follow our conventions"). State the don'ts as loudly as the do's, and point to one existing file as a style example. Keep it under ~40 lines – I'll prune from there.
```

Prompts como un ingeniero



La diferencia entre una sesión frustrante con la IA (Inteligencia Artificial) y una productiva casi nunca es el modelo. Es el prompt (la instrucción que le das). La mayoría de los principiantes tratan a la IA como un buscador: escriben un deseo vago y esperan lo mejor. Los ingenieros con experiencia la tratan como a un compañero junior, agudo pero literal: alguien rápido y capaz, pero que construirá exactamente lo que describas, incluidos tus errores.

Este capítulo trata de cerrar esa brecha. No necesitas memorizar trucos ni acumular una carpeta de plantillas de prompts “mágicas”. Necesitas comunicarte como lo hace un buen ingeniero al encargar una tarea: con contexto, restricciones y una definición clara de “terminado”. Todo lo que sigue es una variación de esa única habilidad.

Da contexto antes de dar la tarea

La IA no puede ver tu proyecto como tú lo ves. No conoce la versión de tu framework (marco de trabajo), la estructura de tus archivos, ni las convenciones que ya has establecido. Cuando te saltas esto, ella adivina, y adivina mal. Peor aún, adivina *con confianza*, así que el resultado parece plausible hasta que choca con tu código real.

Compara estos dos prompts para la misma petición.

```
Add a function to validate emails.
```



```
This is a TypeScript backend using Express and Zod for validation.
We already validate inputs with Zod schemas in src/schemas/.
Add an email validation schema in that style. Emails must be
lowercase, max 254 chars, and reject disposable domains from
the existing BLOCKED_DOMAINS list in src/config.ts.
```

El primer prompt produce código genérico que puede usar una expresión regular que no quieres, con un estilo que no coincide con tu código base. El segundo produce código que puedes pegar directamente. La regla: **indica el stack (conjunto de tecnologías), las convenciones y las piezas existentes que la IA debe reutilizar.**

Un buen prompt apila estas piezas en orden, de abajo hacia arriba: el contexto en la base, la petición real cerca de la cima:

EXAMPLES	one input → output
CONSTRAINTS	what NOT to do
EDGE CASES	empty, invalid, errors
INPUTS / OUTPUTS	types and shapes
GOAL	one clear task
CONTEXT	stack, files, patterns

foundation first ▲

Una lista de verificación mental útil antes de darle a enter: el lenguaje y el framework (con la versión, si importa), los archivos o módulos relevantes, los patrones ya presentes en el proyecto, y cualquier biblioteca que la IA deba preferir o evitar. No necesitas los cuatro puntos siempre, pero si la respuesta cambiaría según alguno de ellos, nómbralo. Ante la duda, pega la firma, el tipo o la configuración reales contra los que el código nuevo debe encajar. Unas pocas líneas de código existente le enseñan a la IA más sobre tus convenciones de lo que jamás lograría un párrafo de descripción.

Especifica entradas, salidas y casos límite

Una tarea no está definida hasta que se definen sus límites. Los ingenieros con experiencia piensan en términos de “qué entra, qué sale y qué pasa cuando algo falla”. Incorpora eso al prompt.

```
Write a function to parse a date string.
```

```
Write a function parseDate(input: string): Date | null.
```

```
Inputs: ISO 8601 strings ("2026-06-14", "2026-06-14T10:30:00Z").
```

```
Output: a Date object, or null if the string is invalid.
```

```
Edge cases to handle:
```

- empty string or whitespace -> null
- valid format but impossible date (2026-02-30) -> null
- trailing garbage ("2026-06-14xyz") -> null

```
Do not throw. Return null for all invalid input.
```

Cuando nombras los casos límite, la IA los maneja. Cuando no lo haces, escribe solo el camino feliz y tú descubres los huecos en producción. La misma disciplina aplica más allá de las funciones puras: para un endpoint (punto de acceso de una API), especifica la respuesta de éxito y las respuestas de error (¿qué código de estado en un correo duplicado? ¿en un cuerpo mal formado?). Para un componente de interfaz, especifica el estado vacío y el estado de carga, no solo el estado con datos. El comportamiento interesante de la mayoría del software vive en sus bordes, así que ahí es donde tu prompt debe invertir sus palabras.

Si no estás seguro de cuáles son los casos límite, pregúntale primero a la IA: “¿Qué casos límite debería manejar un analizador de fechas?” — y luego devuelve esa respuesta como restricciones. Esto convierte al modelo de generador de código en generador de listas de verificación, y revisar una lista es más rápido que depurar los casos que olvidaste.

Trabaja en pasos pequeños

El error más común, con diferencia, es pedir demasiado de una sola vez. “Constrúyeme un panel de usuario con autenticación, gráficos y una página de configuración” producirá un muro de código que no puedes revisar, con errores repartidos por archivos que no leíste.

Los pasos pequeños te mantienen en control:

- Entrega una función, componente o endpoint por prompt.
- Verifica que cada pieza funcione antes de seguir.
- Construye sobre código confirmado y funcional, no sobre una torre de resultados sin verificar.

```
Build the whole authentication system with login, signup, password reset, sessions, and email verification.
```

```
Step 1 of the auth flow: write just the signup endpoint. POST /signup taking { email, password }. Hash the password with bcrypt, store the user, return 201 with the user id. Assume the User model and db client already exist (I'll show you their shapes). We'll handle login next.
```

Prompts más pequeños significan diffs (diferencias de código) más pequeños, y diffs más pequeños significan que realmente puedes leer lo que cambió. Ese es todo el juego. Hay una segunda ganancia: cuando algo se rompe, un paso pequeño es trivial de aislar. Si generaste y verificaste el endpoint de registro, luego la lógica de sesión, luego el restablecimiento de contraseña, un error que aparece después del paso tres casi con certeza vive en el paso tres. La generación de golpe grande destruye esa señal: el error podría estar en cualquiera de cuatrocientas líneas que nunca leíste.

Pide un plan antes del código

Para cualquier cosa no trivial, separa *pensar* de *escribir*. Pídele a la IA que esboce su enfoque primero. Esto atrapa suposiciones erróneas antes de que se conviertan en 200 líneas de código equivocado, y es mucho más barato corregir un plan que un pull request (solicitud de incorporación de cambios).

```
I want to add rate limiting to my API. Before writing any code, give me a short plan: where the limiter should live, what storage it needs, what happens when a limit is hit, and any tradeoffs. Don't write code yet — I'll approve the approach first.
```

Cuando el plan se ve bien, di “bien, implementa el paso 1”. Cuando no, has gastado treinta segundos corrigiendo el rumbo en lugar de una hora desenredando código. Planificar primero es el hábito que más separa el prompting al estilo ingeniero del prompting basado en deseos.

La etapa del plan es también donde detectas que la IA elige una solución más pesada de la que necesitas: recurrir a Redis cuando bastaría un mapa en memoria, o construir a mano algo que una llamada a una

biblioteca de una sola línea ya resuelve. Lee el plan con una pregunta en mente: *¿es esto lo más simple que podría funcionar?* Si el enfoque huele a sobreingeniería, objeta con palabras simples (“este es un proyecto personal con un solo servidor, ¿podemos saltarnos el almacén distribuido?”) antes de que se escriba una sola línea.

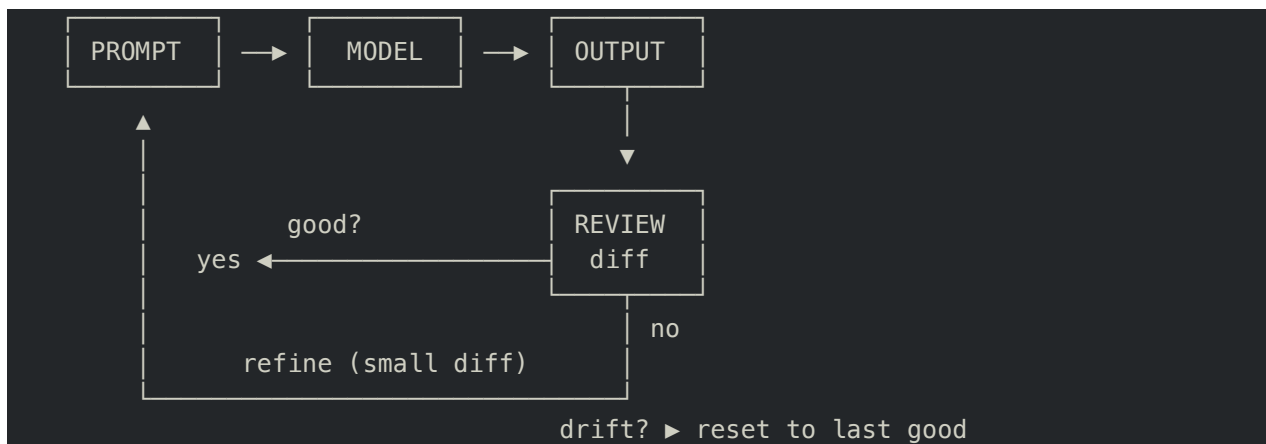
Itera sobre diffs, no reescrituras

Una vez que el código existe, resiste la tentación de pedir todo de nuevo. Seguimientos vagos como “hazlo mejor” descartan código que funciona y reintroducen errores que ya habías corregido. Señala el cambio específico que quieres.

```
That's not right, rewrite the whole thing.
```

```
The function works, but two issues:  
1. The error on line with `throw new Error` should return a  
   Result type instead of throwing – match the pattern in  
   the validateUser function above.  
2. The loop re-reads `items.length` each iteration; hoist it.  
  
Show me just the diff for those two changes.
```

El ciclo saludable es un bucle ajustado: dale el prompt al modelo, lee el resultado, luego refina con un diff pequeño y dirigido —nunca una reescritura completa— hasta que quede bien:



Pedir un diff (en lugar de una reescritura completa) mantiene los cambios revisables y preserva las partes que ya funcionan. Trata el resultado de la IA como el pull request de un colega: comenta líneas específicas, pide ediciones dirigidas. Cuando un cambio se descarrila y el código se aleja cada vez más de lo que querías con cada respuesta, no sigas parcheando: vuelve a la última versión en la que confiabas y vuelve a dar el prompt desde ahí con una descripción más precisa. Iterar hacia adelante sobre una base rota solo agrava el desorden.

Proporciona ejemplos

Un ejemplo concreto vale más que un párrafo de descripción. Si quieres un resultado con una forma particular, muestra la forma. Esta es la manera más rápida de eliminar la ambigüedad.

```
Format the data nicely.
```

```
Transform this array of users into a summary object.
```

```
Input:
```

```
[{ name: "Ann", role: "admin" }, { name: "Bo", role: "user" }]
```

Desired output:

```
{ total: 2, byRole: { admin: ["Ann"], user: ["Bo"] } }
```

Write the transform function and a test matching this example.

Los ejemplos fijan la nomenclatura, la estructura y el formato de mayúsculas/minúsculas que las palabras solas dejan borrosos. Cuando puedas, incluye el ejemplo como una prueba (test); así “haz que la prueba pase” se convierte en una instrucción inequívoca. El truco se generaliza: una respuesta de muestra de una API (Application Programming Interface — Interfaz de Programación de Aplicaciones, la forma en que un programa solicita datos a otro) fija un analizador, un par antes/después fija una refactorización, una captura de pantalla o un boceto en ASCII (código estándar de caracteres) fija un diseño. Cada vez que te sorprendas escribiendo una larga descripción de una forma, detente y muestra una instancia de esa forma en su lugar.

Dile a la IA qué NO hacer

Las restricciones no son solo sobre el resultado que quieres, también son sobre los movimientos que quieres que la IA evite. Sin restricciones, los modelos tienden a agregar dependencias, a refactorizar “útilmente” código que no pediste que tocaran, y a inventar funciones alrededor de los bordes de tu petición. Unas pocas restricciones negativas mantienen el resultado ajustado.

```
Add a debounce to the search input.
```

```
Add a debounce to the search input.
```

Constraints:

- No new dependencies — we already have lodash, use that.
- Touch only SearchBar.tsx. Do not refactor the parent.
- Keep the existing prop names and types unchanged.
- 300ms delay.

El segundo prompt bloquea de antemano una docena de formas en que el primero podría salir mal. Las restricciones negativas son especialmente valiosas en un código base existente, donde el costo de un cambio no solicitado no es solo revisarlo: es el conflicto de fusión, el compañero de equipo sorprendido y la regresión en código que funcionaba bien. Si hay una línea que no quieres que se cruce, dilo explícitamente; la IA no va a inferir tus límites por cortesía.

Administra el contexto como un recurso

Una conversación larga no es gratuita. A medida que un hilo crece, las instrucciones anteriores quedan enterradas, el modelo se apoya más en los mensajes recientes, y los detalles que indicaste hace veinte turnos silenciosamente dejan de importar. Imagina la ventana de contexto (el espacio de texto que el modelo puede “ver” a la vez) como una caja de tamaño fijo: a medida que la conversación crece se va llenando, y tus primeras instrucciones son empujadas hacia el fondo, donde el modelo les presta la menor atención:

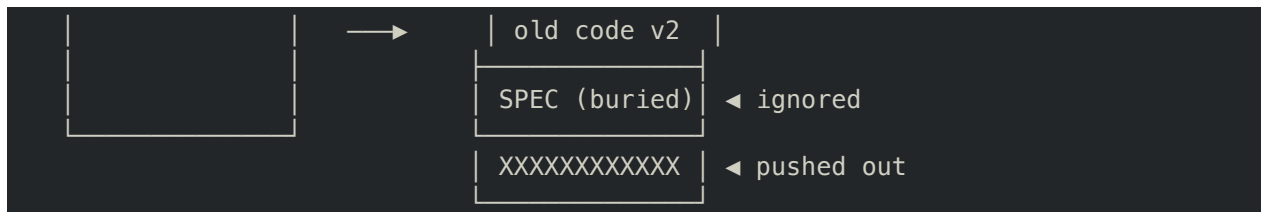
early turn

```
SPEC (clear)
```

long conversation

```
chatter  
chatter  
fix attempt  
old code v3
```

◀ recent (heeded)



Trata la ventana de contexto como una superficie de trabajo de la que eres responsable de mantener limpia, no como una memoria infinita.

Unos cuantos hábitos ayudan:

- **Reformula la especificación cuando cambies de rumbo.** No asumas que un requisito de hace diez mensajes todavía es válido. Resume el objetivo actual en un párrafo corto y deja que la IA trabaje a partir de ahí.
- **Empieza de cero para un problema nuevo.** Cuando pases del código de autenticación a un error de CSS (Cascading Style Sheets — Hojas de Estilo en Cascada, el lenguaje que controla cómo se ve una página), abre una conversación nueva. Arrastrar contexto no relacionado hace que el modelo empeore, no que mejore.
- **Pega el código actual, no el historial.** Si una función ha cambiado cinco veces, la IA no necesita las cinco versiones anteriores, necesita la que existe ahora. Dale el estado presente y el siguiente cambio.
- **Mantén una fuente breve de la verdad fuera del chat.** Para una tarea más larga, unas pocas líneas en un archivo (“stack, decisiones clave, qué está hecho, qué sigue”) te permiten re-sembrar una conversación nueva en segundos en lugar de reconstruirla desplazándote hacia atrás.

El objetivo es el mismo que recorre todo este capítulo: en cada momento, la IA debería estar trabajando a partir de una imagen precisa y compacta de la tarea, no de una transcripción extensa por la que tiene que adivinar su camino.

Antipatrones comunes que evitar

Estos son los hábitos de prompting que silenciosamente desperdician tu tiempo. Vigílalos en tus propias sesiones.

- **El deseo vago.** “Haz una buena página de inicio de sesión.” ¿Buena según qué criterio? Reemplaza los adjetivos por especificaciones: qué campos, qué validación, qué framework.
- **El megaprompt.** Volcar una función completa en un solo mensaje. Divídela en pasos que puedas verificar de uno en uno.
- **El contexto silencioso.** Asumir que la IA recuerda o conoce tu proyecto. Reformula los archivos, tipos y convenciones relevantes, especialmente en una conversación nueva.
- **La aceptación ciega.** Pegar código generado sin leerlo. La IA suena segura incluso cuando se equivoca; tú sigues siendo el ingeniero responsable.
- **El reintento infinito.** Volver a ejecutar “inténtalo de nuevo” esperando un resultado distinto. Si dos intentos fallan, el problema es el prompt: agrega contexto o restricciones, no vuelvas a tirar los dados.
- **El objetivo móvil.** Cambiar los requisitos a mitad del hilo sin reformularlos. Resume la especificación actual cuando cambies de rumbo, para que la IA no esté trabajando con una imagen desactualizada.
- **El bucle de halagos.** Aceptar “¡Tienes toda la razón!” como prueba de que la corrección es correcta. Estar de acuerdo no es verificar: ejecuta el código y comprueba tú mismo el comportamiento.

La mentalidad central

Cada técnica aquí se reduce a una sola idea: **di lo que tendrías que decirle a un compañero de equipo competente que no puede leerte la mente.** Contexto por adelantado. Entradas y salidas claras. Pasos pequeños y verificables. Un plan antes del código. Ediciones dirigidas en vez de reescrituras. Ejemplos

donde las palabras se quedan cortas. Límites explícitos sobre qué no tocar. Una imagen limpia y actual de la tarea en todo momento.

No estás escribiendo hechizos mágicos. Estás escribiendo una especificación, breve y precisa, y luego revisando el resultado. Haz eso con constancia y la IA deja de sentirse como una máquina tragamonedas y empieza a sentirse como el ingeniero rápido e incansable que realmente puede ser, y, tan importante como eso, tú sigues siendo la persona que entiende lo que se envió a producción.

Resumen y práctica

Puntos clave

- Da contexto antes de la tarea: indica el stack, las convenciones y las piezas existentes a reutilizar, o pega la firma real contra la que el código debe encajar.
- Define los límites: entradas, salidas y casos límite. El comportamiento interesante del software vive en sus bordes, así que invierte ahí tus palabras.
- Trabaja en pasos pequeños y verificables, y pide un plan antes del código; es mucho más barato corregir un plan que un pull request.
- Itera sobre diffs dirigidos, no reescrituras completas, y usa ejemplos concretos (idealmente como pruebas) para fijar todo lo que las palabras dejan borroso.
- Administra la ventana de contexto como un recurso, y di qué no hacer: las restricciones negativas explícitas mantienen el resultado ajustado.

Ponlo en práctica

Toma una petición vaga de una sola línea que normalmente escribirías (“agregar búsqueda a la lista”) y reescríbela como un prompt de ingeniero antes de enviar nada. Agrega el contexto (stack, el archivo que toca), las entradas/salidas, dos o tres casos límite, un ejemplo del resultado deseado, y una restricción negativa (qué no tocar). Envía la versión reescrita y compárala con lo que habría producido la versión perezosa. Haz esto con tres peticiones reales y la mejora se vuelve automática.

Prompt del capítulo

```
Context: <language + framework + version>, working in <file/module>.  
We already <relevant existing pattern or library to reuse>.
```

```
Task: <one clear thing to build>.
```

```
Inputs / outputs: <types and shapes>.
```

```
Edge cases to handle: <list the ones that matter>.
```

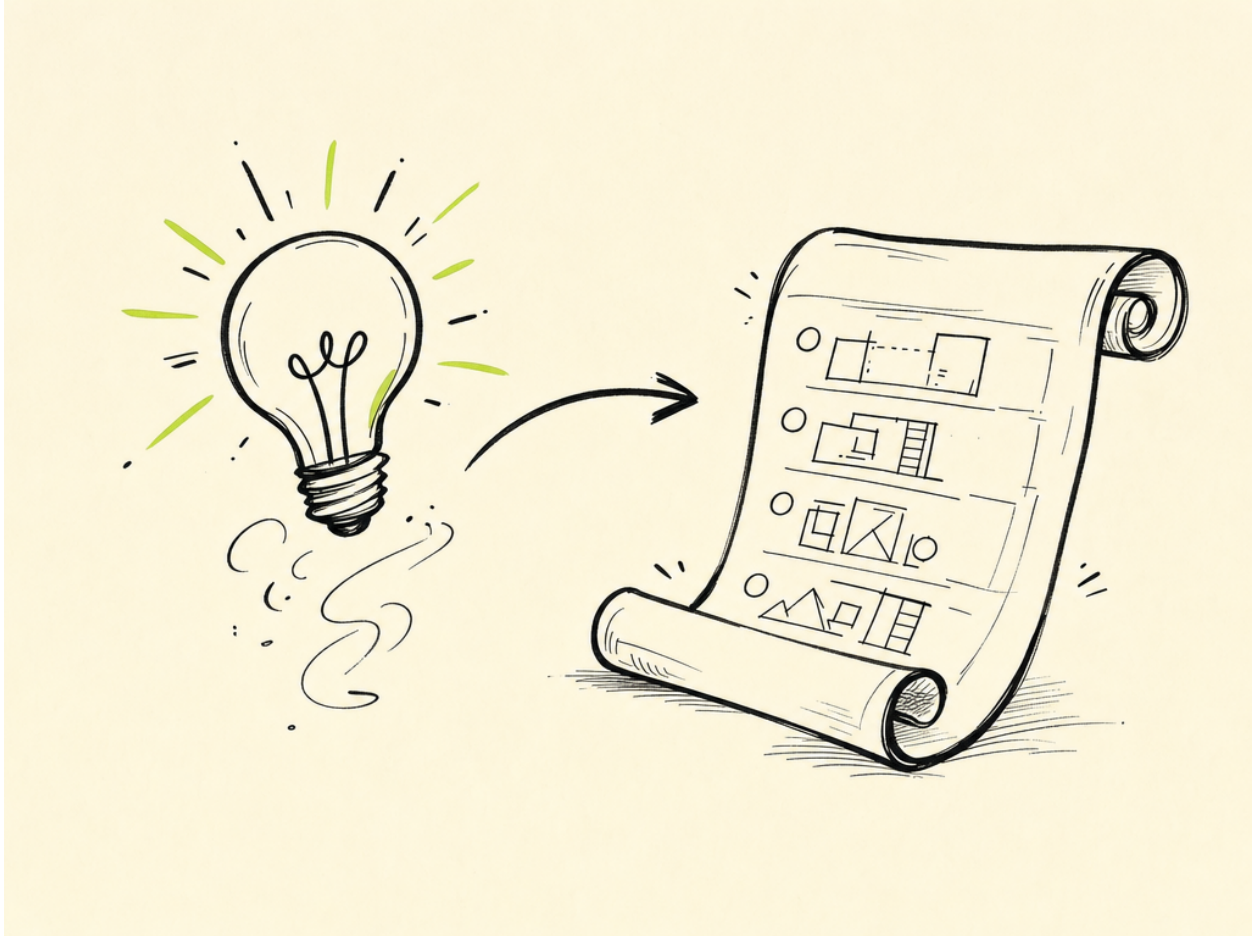
```
Example: <one concrete input -> expected output>.
```

```
Constraints:
```

- No new dependencies – use <what we already have>.
- Touch only <this file>; don't refactor anything else.
- Keep existing names and signatures unchanged.

```
Before coding, give me a 3-4 line plan and wait for my "go".
```


De la idea a la especificación en minutos



Todo proyecto comienza como una idea difusa: “quiero una app que me ayude a llevar el control de mis lecturas”. Es una buena semilla, pero no puedes construir directamente a partir de ella — y tu IA tampoco. Una idea vaga entregada a una IA produce software vago: funciones que no pediste, piezas que sí necesitabas pero faltan, y un montón de código que no entiendes.

La solución es una **especificación** (o *spec*) — un documento breve y en lenguaje sencillo que dice qué vas a construir y qué *no* vas a construir. La buena noticia: con la IA como compañera de pensamiento, puedes pasar de una idea de una línea a una especificación lista para construir en minutos, no en días. Este capítulo te muestra cómo.

Para que quede claro qué *no* es una especificación: no es un contrato, no es un documento de diseño que vayas a defender en una reunión, y no es algo que escribes una vez y congelas. Es una nota de trabajo para ti y para tu IA. La irás editando a medida que aprendas. El objetivo no es un documento perfecto — es uno *suficientemente bueno* que haga más claras las próximas horas de construcción que las anteriores.

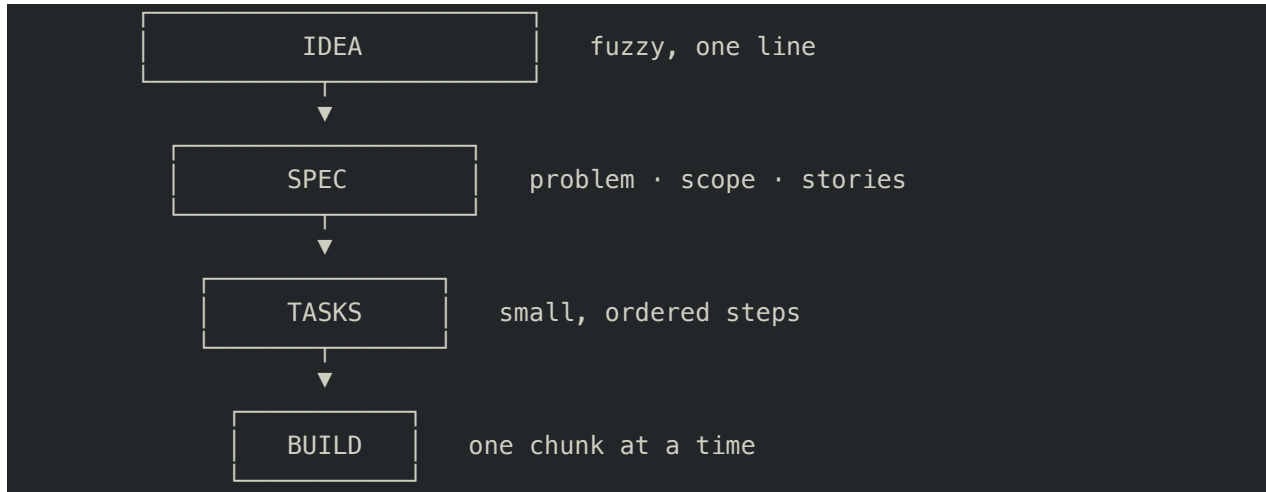
Por qué una especificación, y no solo intuición

Programar por intuición (*vibe coding*) no significa saltarse la planificación. Significa planificar *rápido* y *ligero*, y luego dirigir a la IA para que ejecute. Una especificación es el volante. Sin ella, la IA adivina tu intención en cada instrucción, y los pequeños malentendidos se acumulan hasta convertirse en un caos.

Una especificación te da tres cosas:

- **Contexto compartido** — puedes pegarla en cualquier sesión de IA y restablecer al instante qué estás construyendo.
- **Un límite de alcance** — una línea escrita entre lo que “sí” y lo que “no”, que frena la expansión descontrolada de funciones.
- **Una lista de tareas** — fragmentos construibles que la IA puede completar uno a la vez.

Una especificación funciona como un embudo: toma una idea amplia y difusa y la va reduciendo, paso a paso, hasta convertirla en pequeños fragmentos que la IA realmente puede construir.



No necesita ser larga. Una página es suficiente para la mayoría de los proyectos. De hecho, una especificación que crece más allá de dos páginas suele ser una señal de alerta: o estás planificando demasiado lejos en el futuro, o estás ampliando el alcance sin darte cuenta. Ante la duda, recorta. Las partes que eliminas rara vez se extrañan, y las partes que conservas se vuelven más nítidas.

Paso 1: Aclarar el problema

Antes de pensar en funciones, define el problema y la persona que lo tiene. Resiste la tentación de saltar directo a las soluciones. Un enunciado de problema claro mantiene honesta cada decisión posterior.

Pregúntate a ti mismo (o pídele a la IA que te entreviste):

- ¿Para quién es esto? (Sé específico — “yo”, “creadores independientes”, “mi club de running”).
- ¿Qué están haciendo hoy, de forma dolorosa, sin esto?
- ¿Cómo se ve el éxito en una sola frase?

Puedes dejar que la IA haga la excavación. Prueba esta instrucción:

```

I have a rough idea: a web app to track books I'm reading.
Act as a product thinking partner. Ask me 5 sharp questions,
one at a time, to clarify the problem, the target user, and what
"done" looks like. Don't propose features yet – just interrogate
the idea until it's concrete.
  
```

Responder cinco buenas preguntas suele afinar una idea vaga hasta convertirla en algo real. El truco está en dejar que la IA te haga contraargumentos en lugar de darte la razón en todo. Si pregunta “¿necesitas sincronizar entre dispositivos?” y tu respuesta honesta es “no, solo mi laptop”, acabas de eliminar un backend, un sistema de autenticación y semanas de trabajo — antes de escribir una sola línea de código. Ese es el punto: el lugar más barato para recortar una función es en una conversación, no en un código ya escrito.

Observa el momento en que tus respuestas dejan de ser “eh, supongo que sí” y pasan a ser “sí, exactamente eso”. Ese cambio es la señal de que has encontrado el problema real y puedes avanzar.

Paso 2: Definir el alcance y los no objetivos

Este es el paso más valioso y el que más se salta la gente. Los **no objetivos** — las cosas que deliberadamente *no* vas a construir — te protegen de una IA a la que le encanta añadir extras “útiles”.

Para nuestro rastreador de libros, el alcance podría ser:

- **Dentro del alcance:** añadir un libro, marcarlo como leyendo/terminado, ver una lista, calificar los libros terminados.
- **Fuera del alcance (no objetivos):** funciones sociales, recomendaciones, importar desde Goodreads, app móvil, cuentas de usuario.

Imagina dos columnas: todo lo que te comprometes a construir a la izquierda, todo lo que deliberadamente omites a la derecha. La línea entre ambas es el límite que mantiene pequeña la versión 1.

IN SCOPE	NON-GOALS
Add a book Mark reading/done See a list Rate finished books	Social features Recommendations Goodreads import Mobile app User accounts
build now ←————→ not now	

Escribir los no objetivos cumple dos funciones. Mantiene tu primera versión lo bastante pequeña como para realmente terminarla, y te da una frase que puedes pegar cuando la IA se desvía: “*Recuerda, las cuentas de usuario son un no objetivo para la v1 — no añadas login.*”

Un no objetivo no significa “nunca” — significa “ahora no”. Poner las cuentas de usuario en la lista de no objetivos no significa que tu app nunca las vaya a tener. Significa que no forman parte de esta versión, así que no pagas su costo hoy. Este reencuadre importa: te permite decir que no a una función sin sentir que estás abandonando una buena idea. Solo estás ordenando la secuencia. La mayoría de los elementos de tu lista de no objetivos en realidad son una lista de tareas pendientes para la v2 disfrazada, y eso está bien — anótalos en algún lugar y sigue adelante.

Paso 3: Escribir historias de usuario

Las historias de usuario convierten el alcance en comportamiento comprobable. El formato es simple:

Como **[usuario]**, quiero **[acción]**, para **[beneficio]**.

Para el rastreador de libros:

- Como lector, quiero añadir un libro por título y autor, para poder empezar a llevar su seguimiento.
- Como lector, quiero marcar un libro como terminado, para poder ver mi progreso.
- Como lector, quiero calificar un libro terminado de 1 a 5 estrellas, para recordar qué me gustó.

Cada historia es una unidad que puedes construir y verificar de forma independiente. Si no puedes imaginar cómo probarías una historia haciendo clic en la app, es demasiado vaga — divídela o reescribela.

La cláusula “para” es la parte que la gente omite, y es la más importante. “Como lector, quiero una caja de búsqueda” le dice a la IA *qué* construir pero no *por qué*, así que no puede tomar buenas decisiones de compromiso. “Como lector, quiero encontrar un libro en mi lista escribiendo parte de su título, para no tener que desplazarme por docenas de entradas” le dice el trabajo real — y ahora un simple filtro mientras escribes podría superar a un motor de búsqueda pesado. El beneficio es la especificación de la especificación: explica la función lo suficientemente bien como para que la IA (y tu yo futuro) puedan distinguir una buena implementación de una equivocada.

Paso 4: Redactar un PRD ligero

Un PRD (Documento de Requisitos de Producto, por sus siglas en inglés *Product Requirements Document*) es simplemente tus decisiones reunidas en un solo lugar. Tiene un esqueleto predecible — el mismo puñado de secciones cada vez:

```
PRD
├─ Problem      why this exists
├─ User         who it's for
├─ Goal         what it does, in one line
├─ In Scope     what v1 includes
├─ Non-Goals    what v1 skips
├─ User Stories as a __, I want __, so that __
├─ Data Model   the fields you store
└─ Done When    the acceptance test
```

Mantenlo pequeño. Aquí tienes un PRD inicial completo que podrías entregarle a una IA:

```
# Book Tracker – Mini PRD

## Problem
I read a lot but lose track of what I've finished and what I thought of it.

## User
Me (a single user). No accounts needed for v1.

## Goal
A simple web app to add books, track reading status, and rate finished ones.

## In Scope (v1)
- Add a book (title, author)
- Set status: To Read / Reading / Finished
- List all books, filterable by status
- Rate finished books 1–5 stars

## Non-Goals (v1)
- User accounts / login
- Social features, sharing, recommendations
- Importing from other services
- Mobile / native apps

## Tech (keep it simple)
- Single-page web app
- Local browser storage (no backend for v1)

## Done When
I can add a book, change its status, rate it, and see my list
persist after a page refresh.
```

Fíjate en la línea **“Done When”** (“cuándo está terminado”). Esa es tu prueba de aceptación — una forma concreta de saber que has entregado la v1. Inclúyela siempre. Un buen “cuándo está terminado” es algo que puedes *realizar físicamente*: hacer clic aquí, escribir esto, refrescar, ver aquello. Si tu condición de terminado suena como un sentimiento (“cuando esté pulido”, “cuando se sienta bien”), nunca estará terminado, porque no hay nada contra qué comprobarlo. Conviértelo en un guion que podrías entregarle a un desconocido.

No tienes que escribir el PRD a mano. Un movimiento útil es entregarle a la IA tus respuestas de los pasos anteriores y dejar que ella arme el borrador, y luego tú lo editas. El sentido de editar en lugar de aceptar tal cual es que *tú* tienes que entender y estar de acuerdo con cada línea — este documento es la fuente de verdad que seguirás pegando una y otra vez, así que un error aquí se copia en cada sesión posterior.

Paso 5: Esbozar el modelo de datos

La mayoría de las apps giran en realidad en torno a los *datos*: qué almacenas y cómo se relacionan las piezas. Un esbozo rápido del modelo de datos te ahorra código confuso e inconsistente más adelante. No necesitas un diagrama de base de datos — una lista de campos es suficiente.

Para el rastreador de libros, hay una cosa principal — un Book (libro):

```
Book
- id: unique string
- title: string
- author: string
- status: "to_read" | "reading" | "finished"
- rating: number 1-5 (only if finished)
- addedAt: date
```

Incluso este pequeño esbozo hace explícitas las decisiones: las calificaciones solo existen para los libros terminados, el estado es uno de tres valores fijos, y cada libro tiene un id. Cuando le entregas esto a la IA, deja de adivinar los nombres de los campos y construye de forma consistente.

La razón para fijar esto pronto es que los nombres de los campos se filtran por todas partes. En cuanto la IA escribe código que lee `book.rating`, ese nombre aparece en el formulario, en la vista de lista, en la capa de almacenamiento y en cualquier función posterior que añadas. Si dejas que se desvíe — `rating` en un lugar, `stars` en otro, `score` en un tercero — obtienes errores tediosos de rastrear. Un modelo de datos de seis líneas es el seguro más barato contra eso.

Un bloque de especificación breve y listo para copiar y pegar suele funcionar mejor que la prosa para la IA. Aquí está el mismo modelo escrito como una instrucción que realmente enviarías:

```
Use this exact data shape for a book. Don't add fields I didn't list,
and don't rename these:
```

```
Book {
  id      - unique string, generated on creation
  title   - string, required
  author  - string, required
  status  - one of: "to_read" | "reading" | "finished"
  rating  - integer 1-5, present ONLY when status is "finished"
  addedAt - ISO date string, set when the book is added
}
```

```
If you think a field is missing, ask me before adding it.
```

Esa última línea — *pregúntame antes de añadirlo* — está haciendo un trabajo real. Convierte a la IA de adivinadora en colaboradora y mantiene el modelo de datos como tuyo.

Paso 6: Dividir la especificación en tareas del tamaño justo

El último movimiento es partir tu especificación en tareas que la IA pueda ejecutar **una a la vez**. Una tarea del tamaño justo es lo bastante pequeña como para que puedas revisar el resultado en un minuto y saber si funcionó. Si una tarea le toma a la IA varios archivos y no puedes saber si está bien, es demasiado grande.

Buenas tareas para el rastreador de libros:

1. Configurar el diseño básico de la página con un encabezado y una lista de libros vacía.
2. Construir el formulario “añadir un libro” (título + autor) que agrega a la lista en memoria.
3. Añadir estado (Por leer / Leyendo / Terminado) con un menú desplegable para cambiarlo.
4. Guardar la lista en el almacenamiento local del navegador para que sobreviva a un refresco.
5. Añadir un control de calificación que aparezca solo para los libros terminados.
6. Añadir botones de filtro de estado sobre la lista.

Cada tarea corresponde a una historia de usuario o a una línea del PRD, se construye sobre la anterior y termina en algo que puedes *ver* funcionando. Ese es el ritmo de la programación por intuición: tarea pequeña, ejecútala, verifica, siguiente.

Incluso puedes hacer que la IA genere esta desglose por ti:

```
Here is my mini PRD and data model [paste them].  
Break this into 6–8 small, ordered build tasks. Each task should  
be independently testable and build on the previous one. Keep each  
task to something I can verify by clicking around the app.
```

Mantén viva la especificación

La especificación no es un monumento que construyes una vez y admiras. Es un documento vivo, y el error más común es tratarla como fija. A medida que construyas, aprenderás cosas que la etapa de planificación no podía decirte: una función resulta más difícil de lo esperado, un no objetivo termina siendo esencial, una historia de usuario en realidad eran dos historias en secreto. Cuando eso ocurra, edita primero la especificación, y luego dile a la IA que siga la nueva versión. La especificación lidera; el código sigue.

Algunos hábitos la mantienen honesta:

- **Actualízala en el momento en que cambie una decisión.** Una especificación que dice una cosa mientras tu código hace otra es peor que no tener especificación — engaña activamente a la próxima sesión de IA.
- **Tacha las tareas a medida que las terminas.** La lista de tareas funciona también como rastreador de progreso. Ver seis de ocho elementos marcados es en sí mismo un tipo de impulso.
- **Mueve las sorpresas a los no objetivos o a una lista “para después”** en lugar de simplemente hacerlas. Si una función tentadora aparece a mitad de la construcción, anótala y sigue adelante. Puedes decidir más tarde si merece un lugar en la v1.

Esta es la disciplina que separa “construí algo” de “construí lo que quería construir”. La especificación es cómo te mantienes como autor de tu proyecto en lugar de ser un pasajero en él.

Una trampa común: especificar soluciones, no problemas

Vale la pena señalar por separado un modo de fallo, porque es muy fácil caer en él. Cuando te sientas a escribir una especificación, la tentación es describir la *solución* que ya te imaginaste — “una barra lateral con pestañas”, “una ventana modal emergente”, “una tabla de Postgres”. Pero una especificación llena de soluciones bloquea silenciosamente decisiones que aún no te has ganado.

Compara estas dos líneas:

```
Bad: Add a left sidebar with collapsible sections for each book status.  
Good: Let me see my books grouped by status without losing the full list.
```

La primera frase ya ha elegido una barra lateral, secciones plegables y un diseño — antes de que nadie haya confirmado que esas son las decisiones correctas. La segunda enuncia el *trabajo por hacer* y deja a la IA (y a ti) en libertad de encontrar lo más simple que lo satisfaga, que podría ser simplemente tres botones de filtro. Describe el resultado que quieres; deja que la implementación siga siendo negociable el mayor

tiempo posible. Las soluciones son baratas de cambiar en una especificación y costosas de cambiar en código.

La recompensa

Ahora tienes un enunciado del problema, alcance y no objetivos, historias de usuario, un PRD de una página, un modelo de datos y una lista de tareas ordenada — todos los cimientos de un proyecto real, redactados en minutos con la IA como tu caja de resonancia.

Nada de esto requirió que supieras programar. Requirió que supieras lo que querías y que pensaras con claridad durante unos minutos — y la IA se encargó del resto de pensar en voz alta contigo. Ese es el cambio real: todavía no estás escribiendo software, estás escribiendo la *intención*, y una buena intención es la mayor parte de la batalla.

Guarda esta especificación en un archivo dentro de tu proyecto (mucha gente usa un `SPEC.md` o `PRD.md`). Se convierte en el documento que pegas en sesiones de IA nuevas para reanclar el contexto, y en la lista de verificación que vas completando una tarea del tamaño justo a la vez. En el próximo capítulo, tomaremos la primera tarea y la construiremos de verdad.

Repaso y práctica

Puntos clave

- Una especificación es una nota de trabajo breve y en lenguaje sencillo — qué vas a construir y qué *no* — no un contrato que congelas; una página es suficiente.
- Aclara el problema y la persona antes de las funciones, y anota los no objetivos: son tu defensa más afilada contra una IA que añade extras “útiles”.
- Las historias de usuario con una cláusula “para” y un pequeño PRD con una línea concreta de “cuándo está terminado” convierten la intención en comportamiento comprobable y construible.
- Fija pronto el modelo de datos — los nombres de los campos se filtran por todas partes — y divide la especificación en tareas del tamaño justo que puedas verificar una a la vez.
- Describe el problema, no la solución; los resultados siguen siendo baratos de cambiar en una especificación y costosos de cambiar en código. Mantén viva la especificación a medida que aprendes.

Ponlo en práctica

Toma cualquier idea que hayas estado postergando y conviértela en una especificación de una página en una sola sesión de IA. Haz que la IA te entreviste primero con cinco preguntas afiladas, y luego escribe — con tus propias palabras — el problema, el usuario, tres historias de usuario (cada una con su “para”), una lista explícita de no objetivos, un modelo de datos de seis líneas y una línea de “cuándo está terminado” que puedas realizar físicamente. Guárdala como `SPEC.md`. La prueba de una buena especificación: podrías pegarla en un chat completamente nuevo mañana y la IA sabría exactamente qué construir.

Instrucción del capítulo

```
I have a rough idea: <one-line description>.
```

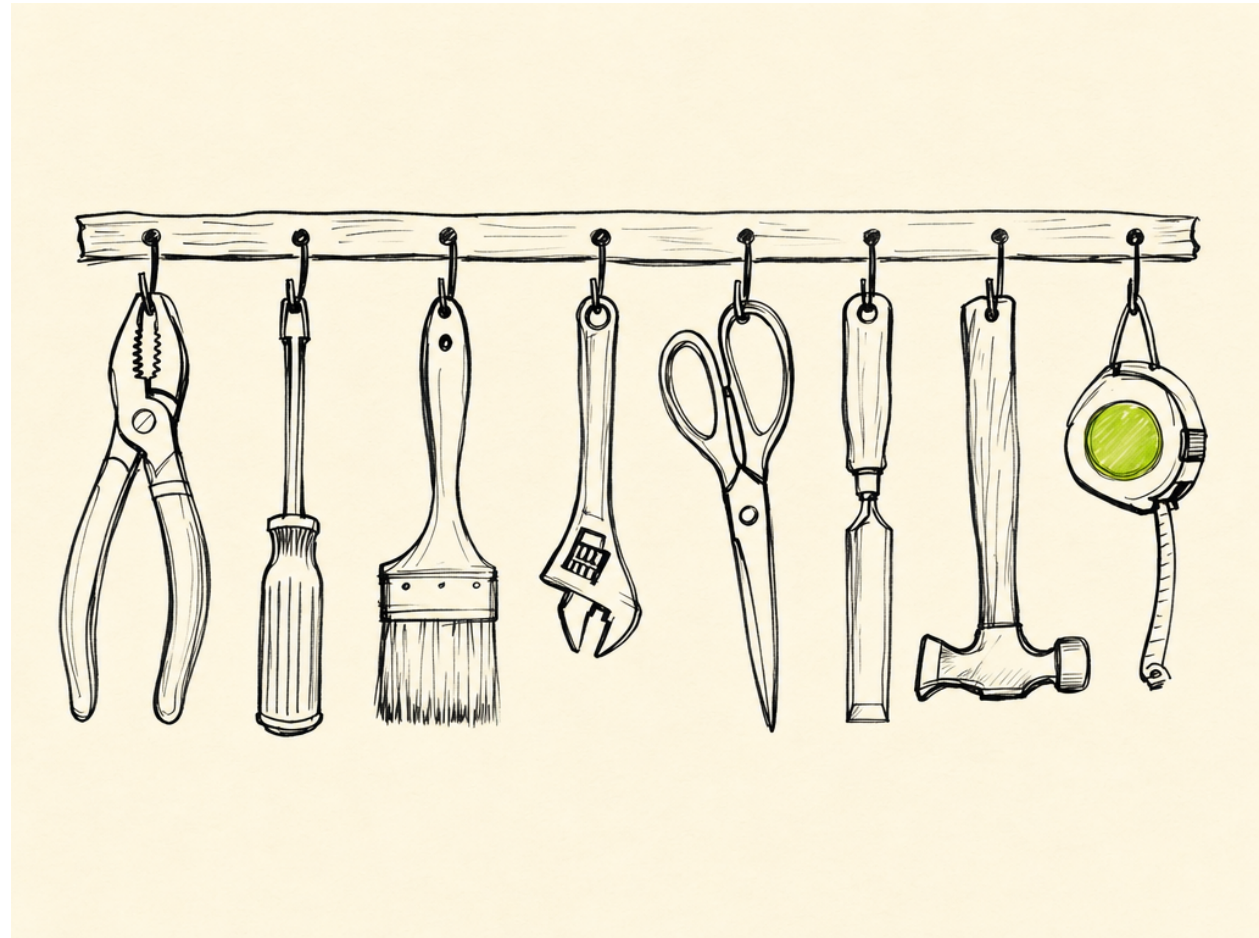
```
Act as my product thinking partner. First, ask me 5 sharp questions, one at a time, to clarify the problem, the target user, and what "done" looks like. Don't propose features yet.
```

```
After I answer, draft a one-page spec with these sections:
```

- Problem – User – Goal
- In Scope (v1) – Non-Goals (v1)
- User stories (each "As a __, I want __, so that __")
- Data model (a short list of fields)
- Done When (a concrete, physically-performable acceptance test)

Keep it tight – under one page. If something is ambiguous, ask before assuming. I'll edit the draft until I agree with every line.

Eligiendo tu stack: los lenguajes de un vistazo



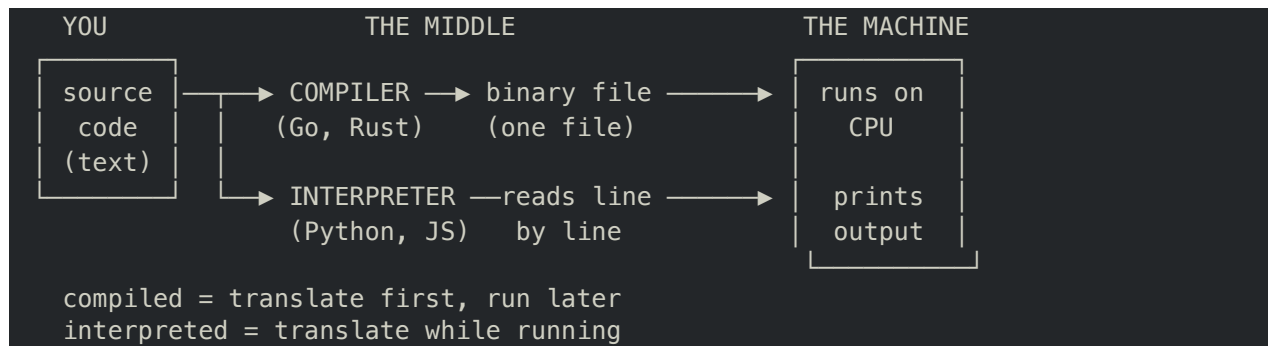
Cuando le pides a una IA que construya software, el lenguaje que elijas sigue importando — y mucho. La IA escribe el código, pero tú vives con las consecuencias: qué tan rápido lanzas, qué tan fácil es desplegar, con qué frecuencia chocas contra un muro. La buena noticia es que no necesitas dominar ninguno de estos lenguajes. Necesitas saber lo suficiente para tomar una decisión sensata y reconocer cuándo la IA te está llevando a un terreno doloroso.

Este capítulo recorre los lenguajes que vale la pena conocer, en qué es genuinamente bueno cada uno, dónde muerde, y — crucial para el *vibe coding* — qué tan bien lo manejan los asistentes de IA. Terminaremos con una guía sencillísima para elegir según tu objetivo.

Una nota antes de empezar: los asistentes de IA son *mucho* mejores en lenguajes populares con montañas de código público que en los nicho. Este no es un efecto pequeño. A menudo es el factor individual más grande en si tu proyecto avanza sin problemas. La popularidad es una característica. Cuando un lenguaje y sus frameworks principales aparecen millones de veces en repositorios públicos, la IA ha visto cada patrón común, cada error típico y cada solución idiomática. Cuando no aparecen, tú pagas esa escasez en intentos fallidos y sinsentidos con tono de seguridad.

Antes de los nombres de los lenguajes, una imagen que vale la pena tener en mente: el código que escribes

nunca es lo que la máquina realmente ejecuta. Algo en el medio convierte tu texto en software en ejecución — ya sea un *compilador* (traduce todo por adelantado a un archivo que ejecutas) o un *intérprete* (lee y ejecuta línea por línea). El lenguaje que elijas determina en qué camino estás.



JavaScript / TypeScript

Este es el predeterminado para la mayoría del vibe coding, y con buena razón. Se ejecuta en todas partes — navegadores, servidores (Node.js), funciones edge — y es el lenguaje de todo el frontend web. TypeScript es JavaScript con un sistema de tipos añadido, y casi siempre deberías preferir TypeScript: los tipos detectan errores temprano y, algo importante, le dan a la IA barandillas que hacen su salida más confiable. La firma de una función tipada es un contrato que el asistente puede leer; sin ella, la IA está adivinando qué forma tienen tus datos, y adivinar es donde se esconden los bugs.

- **Bueno para:** aplicaciones web (frontend y backend), cualquier cosa en la que un usuario haga clic en un navegador, prototipos rápidos, proyectos full-stack en un solo lenguaje, despliegue serverless/edge.
- **Dónde duele:** el ecosistema se mueve rápido y rompe cosas; la proliferación de dependencias es real (una app diminuta puede arrastrar cientos de paquetes); la laxitud del JavaScript plano deja pasar bugs sutiles — otra razón para usar TypeScript.
- **Casos de uso típicos:** paneles de SaaS (Software as a Service — software que alquilas en línea en lugar de instalar), páginas de aterrizaje, comercio electrónico, apps en tiempo real, extensiones de navegador, cualquier cosa que despliegues en Vercel, Netlify o Cloudflare.
- **Manejo por IA:** excelente. Es el lenguaje más representado en los datos de entrenamiento. Los asistentes escriben React, Next.js y Node con confianza y rara vez se atascan. Si no tienes una razón de peso para elegir otra cosa, empieza aquí.

Una compensación concreta que vale la pena conocer: la velocidad que hace de JavaScript algo genial para prototipar es la misma velocidad que lo hace agitarse. Un tutorial de hace dieciocho meses puede ya usar una API obsoleta, y la IA — entrenada con una instantánea del pasado — a veces te entregará el patrón de ayer. Cuando algo que escribe no funciona, la causa suele ser que la librería avanzó. La solución suele ser decirle la versión en la que estás y pedirle que revise la documentación actual. Este es un impuesto que pagas por vivir en el ecosistema más activo del planeta, y para la mayoría de los proyectos vale la pena pagarlo.

Python

Python se lee casi como inglés, lo que lo convierte en el lenguaje más amigable para principiantes y la lengua franca de los datos, la automatización y la propia IA. Si tu proyecto toca aprendizaje automático, análisis de datos, automatización o computación científica, Python suele ser la respuesta.

- **Bueno para:** IA/ML, procesamiento de datos, scripts de automatización, APIs de backend (una API, o Application Programming Interface — interfaz de programación de aplicaciones — es la puerta que un programa usa para hablar con otro), código de pegamento, cualquier cosa donde ya existan librerías (y para Python, suele haberlas).

- **Dónde duele:** lento para cálculo numérico en bruto a menos que te apoyes en librerías escritas en C por dentro; el “infierno de las dependencias” con entornos virtuales y versiones de paquetes confunde a los recién llegados; no sirve para frontends de navegador.
- **Casos de uso típicos:** scraping, pipelines de datos, experimentos de ML, bots de Discord/Telegram, herramientas internas, APIs REST (Representational State Transfer — un estilo común para esas puertas de API) (FastAPI, Django, Flask).
- **Manejo por IA:** excelente, a la par de JavaScript. Los asistentes son extremadamente fluidos en Python y sus librerías principales. Un gran predeterminado para proyectos que no son web.

El único lugar donde Python confiablemente hace tropezar a los recién llegados es el entorno, no el lenguaje. Un script que funciona bien para la IA en una carpeta falla en otra porque los paquetes viven en un entorno virtual que no fue activado, o porque dos proyectos quieren versiones distintas de la misma librería. Nada de esto es culpa de tu código, y nada de esto es difícil una vez que lo has visto — pero es el momento de “por qué no simplemente funciona” más común en Python. Pídele a la IA que configure un entorno virtual y un `requirements.txt` (o que use una herramienta como `uv` o `Poetry`) desde el principio, y la mayor parte del dolor desaparece.

Go

Go fue diseñado en Google para ser simple, rápido y fácil de desplegar. Compila a un único binario auto-contenido que puedes dejar caer en un servidor sin necesidad de instalar ningún runtime. Deliberadamente omite características vistosas, lo que mantiene las bases de código aburridas de la mejor manera — y lo aburrido es fácil de razonar tanto para humanos como para la IA.

- **Bueno para:** servicios de backend rápidos, herramientas de línea de comandos, redes, cualquier cosa que necesite manejar muchas conexiones concurrentes, despliegue sencillo.
- **Dónde duele:** manejo de errores verboso (verás `if err != nil` por todas partes); no es un lenguaje de frontend; la simplicidad deliberada puede sentirse limitante si quieres atajos expresivos.
- **Casos de uso típicos:** APIs, microservicios, CLIs (una CLI, o Command-Line Interface — interfaz de línea de comandos — es una forma de solo texto para ejecutar programas escribiendo comandos), herramientas de infraestructura, backends de alto tráfico.
- **Manejo por IA:** muy bueno. El conjunto de características pequeño y consistente de Go significa que hay una “forma correcta” de hacer la mayoría de las cosas, y los asistentes la siguen bien. Menos datos de entrenamiento que JS/Python, pero la simplicidad lo compensa.

La característica estrella de Go para un constructor solitario es ese binario único. No hay un baile de “instala Node y estos cuarenta paquetes en el servidor”, ni discrepancias de versión de runtime entre tu máquina y producción. Compilas, copias un archivo, lo ejecutas. Eso solo hace que el despliegue sea dramáticamente menos propenso a errores — lo cual importa mucho cuando estás haciendo *vibe coding* y no quieres depurar infraestructura. La verbosidad es el costo: código que sería tres líneas en Python se convierte en ocho en Go. Pero esas líneas extra son predecibles, y predecible es exactamente lo que una IA maneja bien.

Rust

Rust te da velocidad a nivel de C con un compilador que se niega a dejarte escribir categorías enteras de bugs de memoria. La compensación es un compilador famosamente estricto y una curva de aprendizaje empinada. Para el *vibe coding*, Rust es un arma de doble filo: la IA lo escribe, pero la severidad de Rust significa que el compilador rechazará el código con más frecuencia, lo que lleva a más ciclos de ida y vuelta para corregirlo.

- **Bueno para:** software crítico en rendimiento, programación de sistemas, motores de videojuegos, WebAssembly, herramientas de línea de comandos donde la velocidad importa, cualquier cosa donde la corrección y la seguridad sean primordiales.
- **Dónde duele:** curva de aprendizaje empinada; tiempos de compilación lentos; el “borrow checker” frustra a los principiantes; excesivo para la mayoría de las aplicaciones web y scripts.

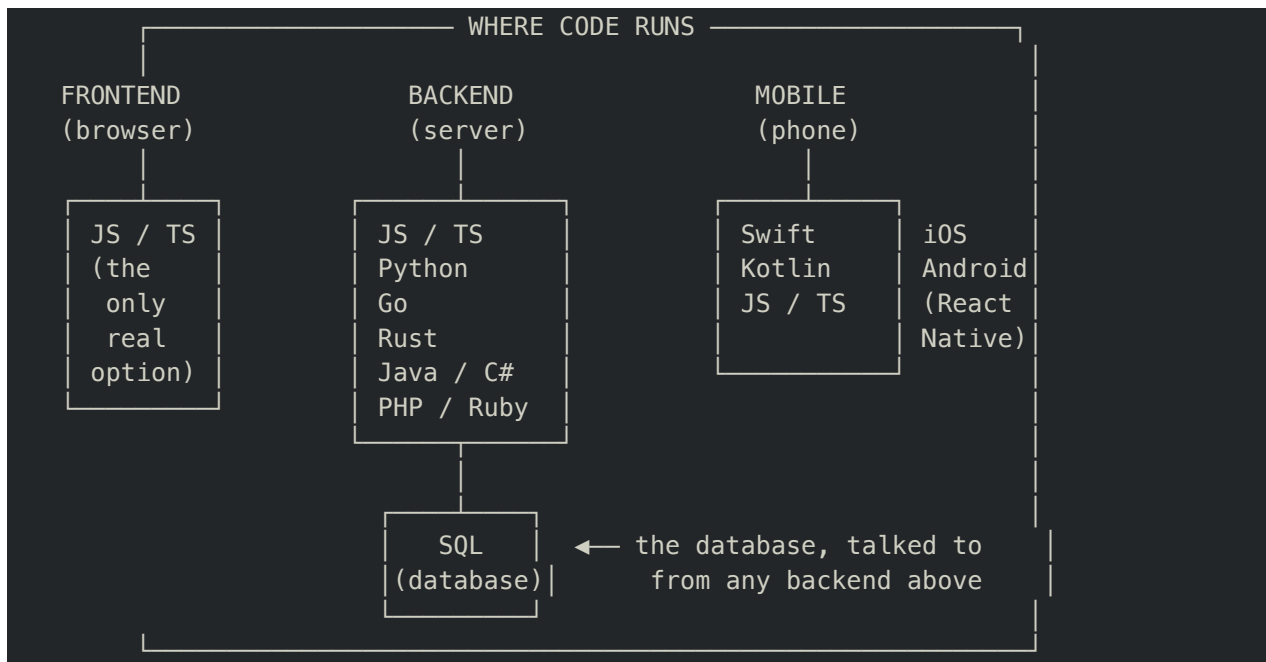
- **Casos de uso típicos:** backends de alto rendimiento, herramientas CLI, módulos WASM (WebAssembly), sistemas cuasi-embebidos, infraestructura donde cada milisegundo cuenta.
- **Manejo por IA:** bueno pero más accidentado. Los asistentes conocen bien Rust, pero el compilador estricto saca a la luz errores que necesitan iteración. Espera más ciclos de “corrige el error, intenta de nuevo” que con Python o JS. Elige Rust cuando el rendimiento o la seguridad justifiquen genuinamente la fricción — no por defecto.

Hay una ventaja sutil en la severidad de Rust que es fácil pasar por alto: el compilador es un revisor gratuito e implacable. Cuando una IA escribe JavaScript descuidado, el bug se envía y muerde a un usuario en tiempo de ejecución. Cuando una IA escribe Rust descuidado, el compilador suele detectarlo antes de que el programa siquiera se ejecute. Los ciclos extra de corrección se sienten como fricción, pero una parte de ellos son el compilador deteniendo un bug real. La pregunta honesta que hay que hacerse antes de recurrir a Rust es si realmente necesitas la velocidad o la seguridad, o si te han dicho que deberías quererlas. Para un primer proyecto, casi nunca las necesitas.

Una palabra rápida sobre los demás

Te toparás con estos, y a veces uno es exactamente el correcto:

Ayuda visualizar dónde tiende a vivir cada lenguaje. El software tiene capas — lo que se ejecuta en el navegador (frontend), lo que se ejecuta en un servidor (backend), lo que se ejecuta en un teléfono (móvil) — y la mayoría de los lenguajes tienen un territorio propio:



- **Java / C#:** los caballos de batalla de las grandes empresas. Maduros, estables, muy bien equipados de herramientas, e ideales para grandes backends corporativos y para Android (Java/Kotlin) o los ecosistemas de Windows/.NET (C#). La IA maneja ambos bien. Como vibre coder solitario, rara vez los *necesitas* a menos que trabajes dentro de un stack empresarial existente o estés construyendo específicamente para Android/Windows.
- **Swift / Kotlin:** los lenguajes nativos para apps de iOS (Swift) y Android (Kotlin). Si estás lanzando una app móvil real a la App Store o a Play Store, probablemente los toques. El soporte de la IA es sólido pero más débil que para los lenguajes web, y el desarrollo móvil conlleva fricción extra (dispositivos, reglas de las tiendas de apps, herramientas de compilación).
- **PHP / Ruby:** la vieja guardia de la web, y todavía ejecuta silenciosamente una porción enorme de ella. PHP impulsa WordPress y, por lo tanto, una gran fracción de todos los sitios web; Ruby on Rails

sigue siendo una forma rápida y con criterio propio de construir una app respaldada por base de datos. Ambos tienen abundantes datos de entrenamiento, así que la IA los maneja bien — pero rara vez son lo primero que un recién llegado elige hoy en día, y típicamente los elegirías porque estás trabajando en una base de código existente en lugar de empezar de cero.

- **SQL (Structured Query Language — lenguaje de consulta estructurado):** no es un lenguaje de propósito general, pero te lo encontrarás en el momento en que tu app almacene datos. Piénsalo como el lenguaje que usas para hacerle preguntas a una base de datos — “dame todos los usuarios que se registraron esta semana”. SQL consulta bases de datos. No eliges SQL *en lugar de* los otros — convive junto a lo que elijas. La buena noticia: la IA es excelente escribiendo SQL, así que puedes describir en palabras simples qué datos quieres y dejar que ella produzca la consulta.

Comparación de un vistazo

Lenguaje	Mejor para	Cuidado con	Fluidez de la IA
JavaScript/TypeScript	Apps web, full-stack, edge	Proliferación de dependencias, agitación rápida	Excelente
Python	IA/ML, datos, scripts, APIs	Cómputo bruto lento, configuración de entorno	Excelente
Go	Backends rápidos, CLIs, servicios	Verboso, no para frontend	Muy buena
Rust	Rendimiento, seguridad, WASM	Curva empinada, más ciclos de corrección	Buena (más accidentada)
Java/C#	Empresa, Android, .NET	Pesado, a menudo excesivo en solitario	Muy buena
Swift/Kotlin	Apps nativas iOS / Android	Fricción móvil, IA más débil	Buena
PHP/Ruby	WordPress, Rails, apps web existentes	Valores predeterminados anticuados, lastre heredado	Muy buena
SQL	Hablar con bases de datos	Convive, no reemplaza	Excelente

El costo real de una elección exótica

Es tentador elegir un lenguaje porque es elegante, o rápido, o porque alguien a quien respetas jura por él. Para el *vibe coding* específicamente, ese instinto puede hundir silenciosamente un proyecto. Aquí está el porqué: cada hora que pasas peleando con las herramientas de un lenguaje de nicho, o desenredando una salida de IA que está sutilmente equivocada porque el modelo ha visto muy pocos ejemplos, es una hora que no pasaste enviando tu producto. La elección exótica rara vez falla de forma escandalosa. Falla como un goteo lento de ciclos de corrección extra, librerías faltantes y respuestas de Stack Overflow que no existen.

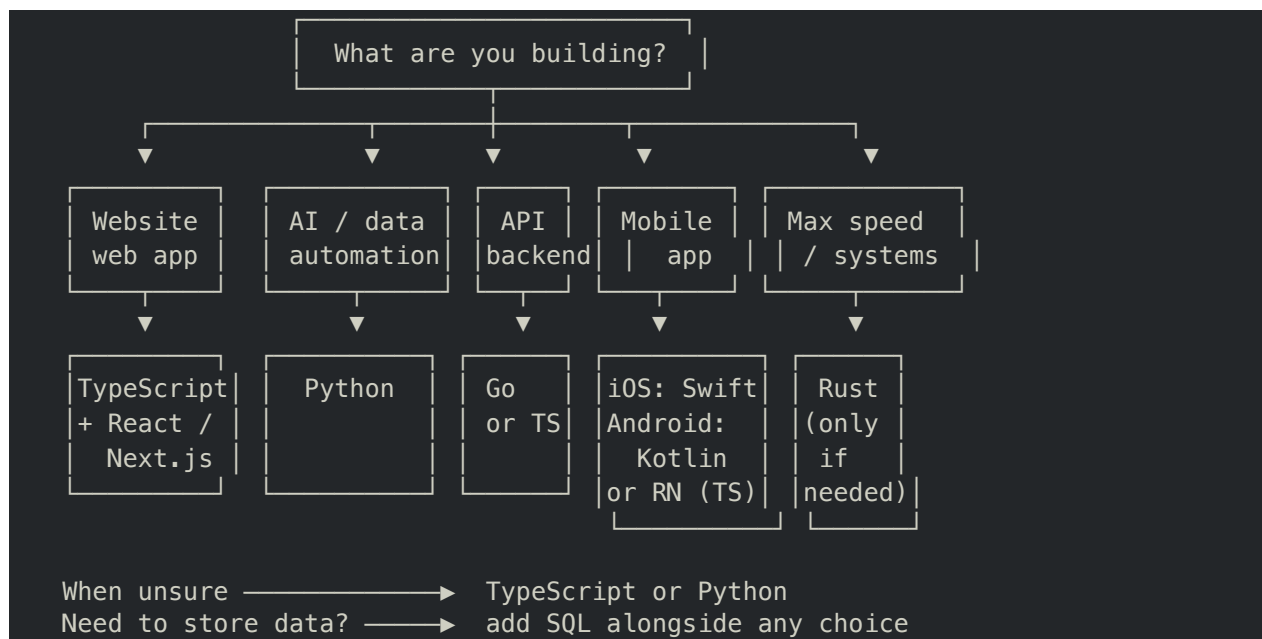
Concretamente, la penalización aparece en tres lugares. Primero, la IA es menos fluida, así que comete más errores y es peor corrigiéndolos — y tú, al no conocer el lenguaje, no puedes distinguir el código bueno del malo. Segundo, el ecosistema es más delgado, así que la librería que necesitas puede no existir, puede estar abandonada, o puede no tener ejemplos que copiar. Tercero, cuando te atascas, la web está más silenciosa: menos tutoriales, menos preguntas respondidas, menos personas que hayan chocado con exactamente tu muro. Con Python o TypeScript, esos tres son no-problemas.

Nada de esto significa que los lenguajes exóticos sean malos. Significa que la carga de la prueba recae en la elección exótica. Si tienes una razón concreta y específica — genuinamente necesitas la velocidad de Rust, la App Store te obliga a usar Swift, estás extendiendo un servicio Go existente — entonces paga el costo con los ojos abiertos. Si la razón es “me pareció cool”, elige la opción popular y lanza algo. Siempre puedes reescribir después, y casi nunca lo harás, porque la elección popular habrá hecho el trabajo.

Elige según tu objetivo

Sáltate la agonía. Empareja tu objetivo con una fila y avanza — siempre puedes cambiar de rumbo después, y para un primer proyecto, el impulso vence a la optimización.

Aquí está toda la decisión como un solo árbol. Empieza arriba, responde cada pregunta y sigue la rama hacia tu stack:



- **Quiero construir un sitio web o una app web.** □ TypeScript (con React/Next.js). La opción obvia y bien respaldada.
- **Quiero hacer IA, datos o automatización.** □ Python. Es donde viven las herramientas.
- **Quiero una API rápida o un servicio de backend.** □ Go (simple y rápido) o TypeScript (si es parte de un stack web). El FastAPI de Python también funciona bien.
- **Estoy construyendo una app móvil.** □ Swift para iOS, Kotlin para Android — o considera un enfoque multiplataforma basado en la web (React Native) para quedarte en TypeScript.
- **Necesito velocidad bruta o control a nivel de sistema.** □ Rust, pero solo si has confirmado que realmente lo necesitas. La mayoría de la gente no lo necesita.
- **Estoy agregando algo a un sitio que ya existe.** □ Lo que sea en lo que ya esté escrito. Un plugin de WordPress significa PHP; una app de Rails antigua significa Ruby. Ajústate a la base de código, no la combatas.
- **Solo quiero aprender y lanzar algo rápido.** □ Python o TypeScript. Ambos son indulgentes, ambos son los lenguajes que la IA conoce mejor, y ambos tienen tutoriales y ejemplos interminables.
- **Mi app necesita almacenar datos.** □ Lo que sea que elijas arriba, más SQL para la base de datos. No eliges entre ellos.

La meta-lección: en caso de duda, elige la opción popular. TypeScript y Python cubren la gran mayoría de lo que un vibe coder solitario alguna vez construirá, y son los lenguajes que tu asistente de IA maneja de forma más confiable. El “mejor” lenguaje para el vibe coding suele ser el que la IA conoce mejor — y eso significa el popular. Guarda las elecciones exóticas para cuando tengas una razón concreta, y deja que la

IA maneje la sintaxis de cualquier manera.

Repaso y práctica

Puntos clave

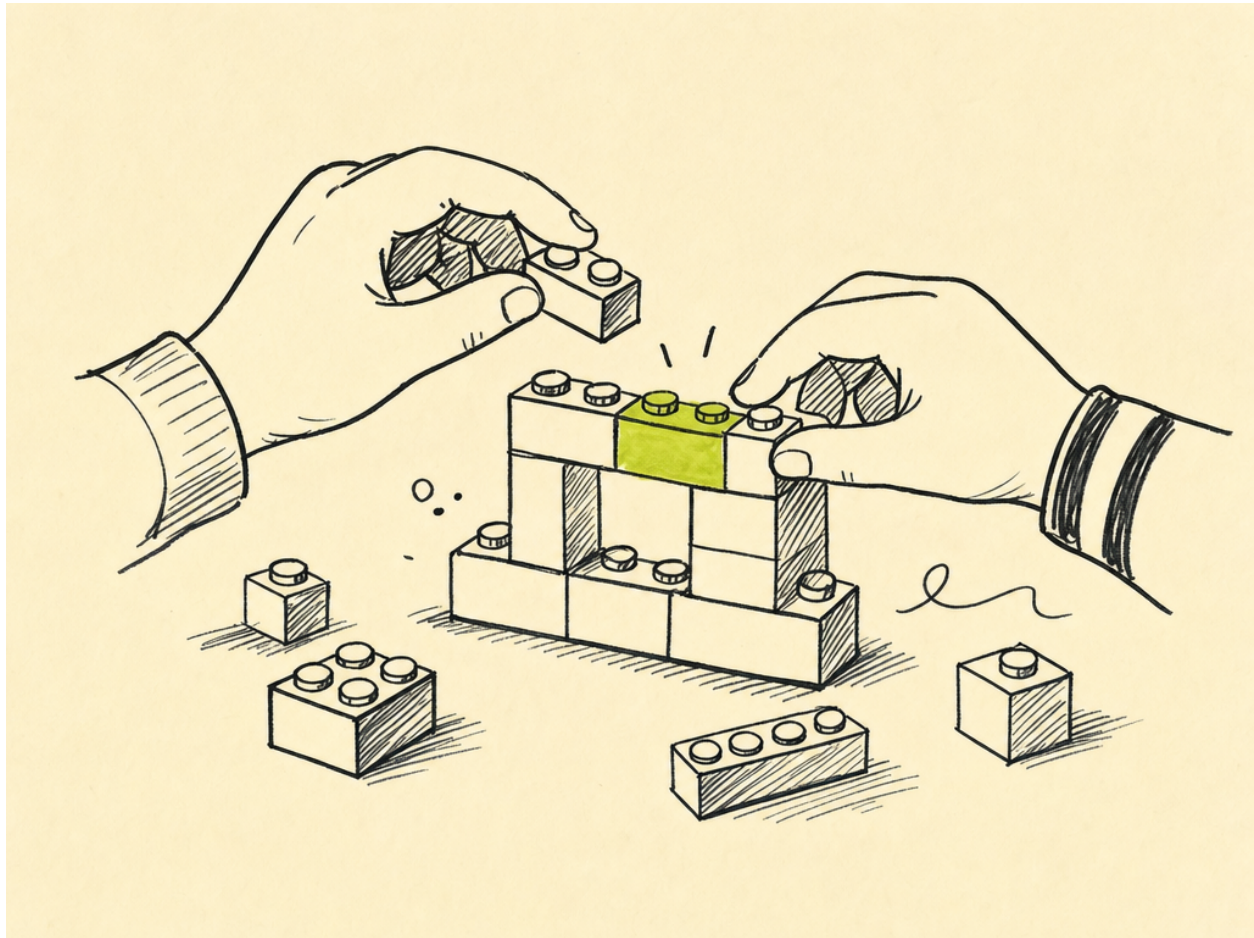
- La IA es mucho más fluida en lenguajes populares que en los de nicho — la popularidad es una característica, a menudo el factor individual más grande en si un proyecto avanza sin problemas.
- TypeScript y Python son los predeterminados seguros: web/full-stack para TypeScript, IA/datos/automatización para Python, y ambos tienen excelente soporte de IA.
- Go (backends de binario único), Rust (velocidad/seguridad, pero más ciclos de corrección), y el resto tienen cada uno un punto óptimo estrecho — recurre a ellos solo con una razón concreta.
- SQL no es una elección de o lo uno o lo otro; convive junto a lo que elijas en el momento en que tu app almacena datos.
- Ajusta el lenguaje a tu objetivo, no a lo que suene impresionante — el impulso vence a la optimización en un primer proyecto.

Práctico

Toma una idea de proyecto que realmente quieras construir y escribe una oración describiendo su objetivo. Empareja con una fila de la lista “Elige según tu objetivo”, luego pídele a tu asistente de IA que confirme o cuestione la elección. Haz que exponga el stack predeterminado (lenguaje, framework, dónde lo desplegarías) en tres viñetas, y pídele que nombre una cosa que haría que un lenguaje *diferente* fuera mejor opción. Terminarás con una decisión de stack justificada antes de escribir una sola línea de código.

```
I want to build the following project: [describe it in one or two sentences].  
Recommend the best language and framework for vibe coding it, optimizing for  
how fluently an AI assistant can write and fix the code, not for raw  
performance. Give me: (1) the language and main framework, (2) where I'd  
deploy it, (3) one realistic gotcha to watch for, and (4) one scenario in  
which a different language would actually be the better choice. Keep it short.
```

Construir funciones, vibe a vibe



A estas alturas ya puedes hablar con tu IA y lograr que escriba código. Pero una función real es más grande que un solo prompt. La habilidad que separa a quienes *lanzan* proyectos de quienes se quedan atascados es saber cómo dividir una función en piezas pequeñas y guiar a la IA a través de ellas una a la vez, manteniendo la aplicación funcionando en cada paso.

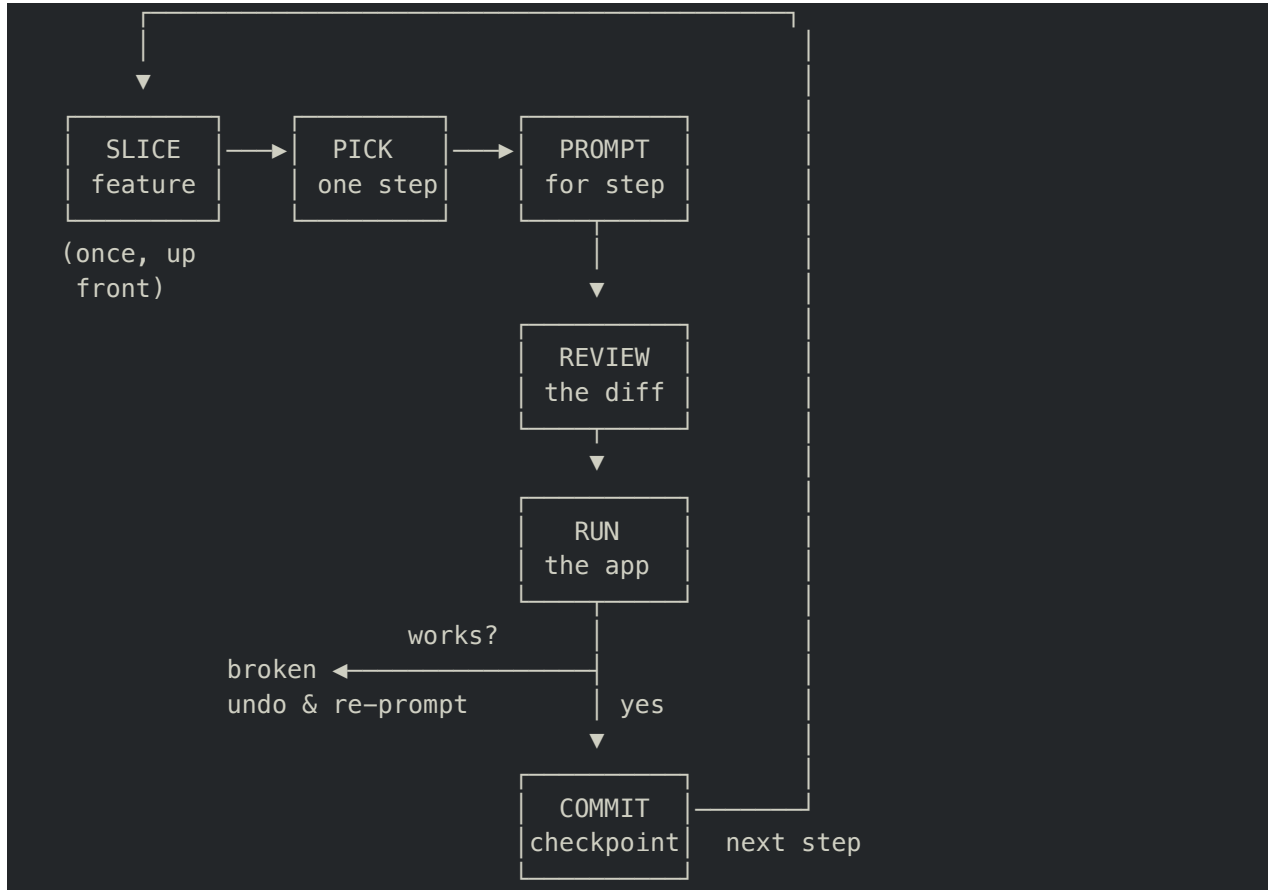
Este capítulo es el ciclo central de construcción. Apréndelo una vez y lo reutilizarás en cada función que construyas de aquí en adelante. Es la diferencia entre un proyecto que crece de forma constante y uno que colapsa bajo su propio peso justo cuando se pone interesante.

El ciclo de construcción

Aquí está el ciclo que ejecutarás una y otra vez. Es pequeño a propósito.

1. **Divide la función en pasos verticales pequeños.** Cada paso debe ser algo que puedas construir, ejecutar y *ver funcionando* en pocos minutos.
2. **Elige un paso.** Solo uno. Ignora todo lo demás por ahora.
3. **Escribe el prompt para ese único paso.** Sé específico sobre qué cambia y qué se mantiene igual.
4. **Revisa el cambio.** Lee lo que escribió la IA. No necesitas entender cada línea, pero échale un vistazo por si hay algo obviamente incorrecto.
5. **Ejecuta la aplicación.** Haz clic en el botón de verdad, carga la página, llama al endpoint. Confirma que el paso funciona.
6. **Confirma (commit).** Guarda un punto de control en el control de versiones con un mensaje breve.
7. **Repite** desde el paso 2 con el siguiente paso, hasta que la función esté terminada.

Dibujado como un ciclo, se ve así: sigues dando vueltas, un paso pequeño por vuelta, hasta que la función está terminada:



Lo esencial es que nunca estás lejos de una aplicación que funciona. Si el paso 4 produce algo roto, solo tienes un pequeño cambio que deshacer, no un enredo gigante.

Quienes se saltan los pasos 4 a 6 se sienten más rápidos durante unos veinte minutos. Después chocan con un error que no pueden ubicar, porque la última versión *conocida como buena* de su aplicación quedó cien cambios atrás. El ciclo se siente lento precisamente porque nunca te deja acumular ese tipo de deuda.

Qué significa “rebanada vertical”

Una **rebanada vertical** (vertical slice) es una porción delgada de la función que atraviesa toda tu aplicación de punta a punta —un poco de interfaz, un poco de lógica, un poco de datos— en lugar de construir una capa entera a la vez.

Enfoque tentador pero equivocado (horizontal):

- Construir todo el esquema de la base de datos.
- Luego construir toda la lógica del backend.
- Luego construir toda la interfaz.

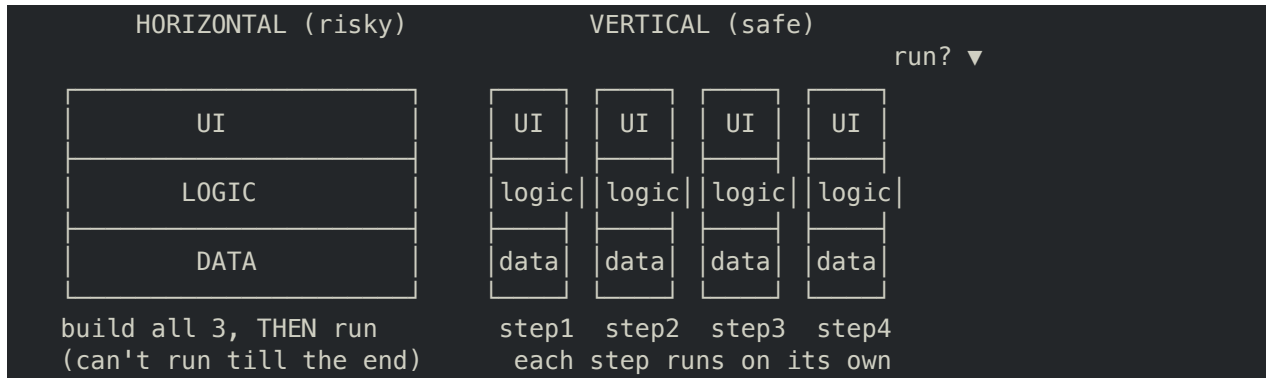
No puedes ejecutar *nada* hasta que las tres estén terminadas, así que no puedes detectar problemas a tiempo.

Mejor enfoque (vertical):

- Paso 1: un botón que hace la versión más simple posible de la función, con datos falsos.
- Paso 2: hacer que use datos reales.

- Paso 3: manejar los casos límite.

La diferencia se ve más fácil lado a lado. El enfoque horizontal construye una capa entera a la vez y no ejecuta nada hasta el final; el vertical construye una franja delgada y ejecutable a través de todas las capas en cada paso:



Cada rebanada es ejecutable. Eso es lo que mantiene la aplicación siempre funcionando.

Una prueba útil para saber si una rebanada es buena: *¿puedes hacer una demo?* Si puedes señalar la pantalla y decir “mira esto”, es una rebanada vertical de verdad. Si la única demo honesta es “bueno, ya está listo el esquema”, construiste una capa horizontal y no sabrás si está bien hasta mucho después, casi siempre en el peor momento posible.

Un ejemplo resuelto: una función de “Favoritos”

Construyamos una pequeña función de principio a fin: los usuarios pueden marcar elementos de una lista como favoritos y ver un contador de cuántos han marcado.

Dividiéndola en rebanadas

Así es como yo la dividiría en pasos verticales:

1. Agregar un botón de estrella junto a cada elemento (todavía no hace nada, solo aparece).
2. Al hacer clic en la estrella, alterna su apariencia encendida/apagada (solo estado, no se guarda).
3. Mostrar un contador “3 favoritos” que refleje los cambios.
4. Guardar los favoritos para que persistan después de recargar la página.

Fíjate que cada paso es ejecutable y visible de forma independiente. Podrías detenerte después de cualquiera de ellos y aun así tener una aplicación funcional.

Paso 1: el botón

Un buen prompt nombra el archivo, el cambio y qué *no* tocar:

In the ItemCard component (src/components/ItemCard.jsx), add a star button in the top-right corner of each card. It should render an outline star icon. Don't wire up any click behavior yet – just make it appear. Keep the rest of the card layout unchanged.

Ejecuta la aplicación. ¿Ves las estrellas? Bien. Confirma el cambio:

```
git add -A && git commit -m "Add star button to item cards"
```

Paso 2: alternar el estado

Make the star button toggle between filled and outline when clicked. Track this with local component state – don't save it anywhere yet. A filled star means favorited.

Piensa en el estado como la memoria de corto plazo de la aplicación: lo que es verdad en este momento, como si una estrella en particular está actualmente rellena. Ahora estás manejando **estado e interfaz juntos**, lo cual es normal. (La interfaz de usuario, o UI, es simplemente la parte de la aplicación que una persona ve y con la que hace clic.) Es probable que la IA agregue algo como esto:

```
const [isFavorite, setIsFavorite] = useState(false);

<button onClick={() => setIsFavorite(!isFavorite)}>
  {isFavorite ? <StarFilled /> : <StarOutline />}
</button>
```

Haz clic en algunas estrellas. ¿Se alternan? Confirma el cambio.

Paso 3: el contador

Add a "★ N favorites" counter at the top of the item list. It should count how many items currently have their star filled. Lift the favorite state up to the list component if you need to so the counter can see it.

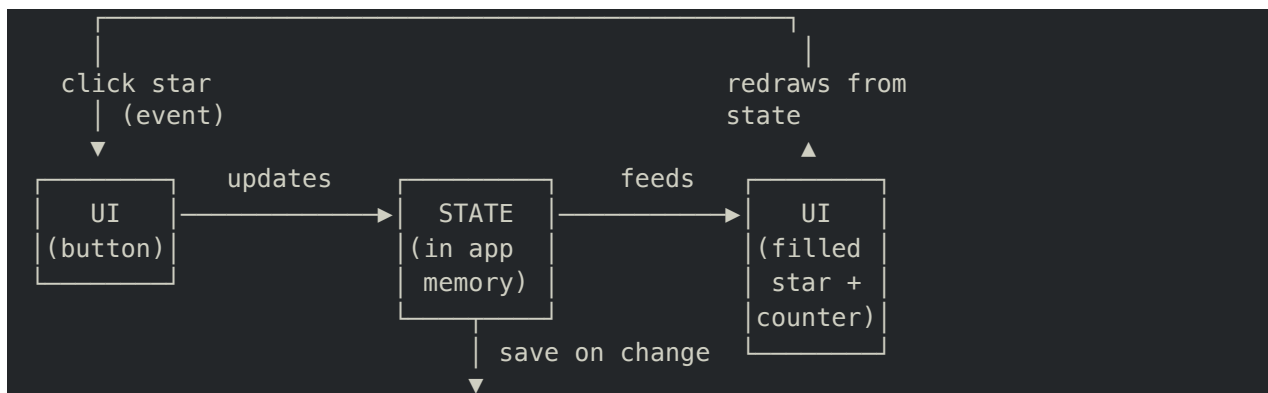
Este paso puede requerir que la IA **reestructure dónde vive el estado**; eso es esperable, y es precisamente por eso que lo haces como un paso pequeño e independiente. Ejecútalo, marca algunas estrellas, observa cómo cambia el número. Confirma el cambio.

Paso 4: hacer que persista

Persist favorites so they survive a page reload. Use localStorage for now – save the list of favorited item IDs whenever it changes, and load it on startup.

(localStorage es solo un pequeño cuaderno de notas que el navegador guarda en la computadora del propio visitante, para que la aplicación pueda anotar cosas y volver a encontrarlas después de una recarga.) Recarga la página. ¿Sigue marcado como favorito? Acabas de lanzar una función, una rebanada ejecutable a la vez.

Así es como se comunican entre sí las piezas de la función terminada. Un clic fluye hacia el estado, la pantalla se vuelve a dibujar a partir del estado, y el estado se refleja en el almacenamiento para que sobreviva a una recarga:




```
STORAGE  
(localSt.)
```

```
← load on startup,  
so it survives reload
```

Leer un diff sin entender cada línea

El paso 4 del ciclo es “revisa el cambio”, y es el que la gente más se salta a medias. No necesitas saber escribir el código tú mismo para revisarlo de forma útil. Necesitas hojearlo buscando un pequeño conjunto de señales de alerta.

Cuando la IA te entrega un cambio, la forma más clara de verlo es el diff: las líneas exactas que se agregaron y se eliminaron:

```
git diff
```

Las líneas que empiezan con + fueron agregadas, las que empiezan con – fueron eliminadas. Léelo como un revisor, no como un compilador. Estás buscando cosas como:

- **¿Tocó archivos que no tenía por qué tocar?** Pediste un contador; ¿por qué aparece `auth.js` en el diff?
- **¿Eliminó algo que parecía importante?** Una línea eliminada (–) que no reconoces vale la pena preguntarla.
- **¿Cambió silenciosamente un comportamiento que no pediste?** Nuevos valores predeterminados, condiciones modificadas, extras “útiles”.
- **¿La magnitud coincide con lo pedido?** Una solicitud de una sola oración que produjo doscientas líneas merece una segunda mirada.

Si algo se ve raro, no tienes que arreglarlo tú mismo; solo pregunta: “¿Por qué cambiaste `auth.js`? Solo pedí el contador.” La mitad de las veces la IA dirá “buena observación” y lo deshará. Ese único hábito atrapa más errores que cualquier cantidad de tiempo mirando fijamente código que no entiendes del todo.

Cuándo dejar que la IA avance libremente y cuándo dirigirla manualmente

No todos los pasos necesitan el mismo control sobre el volante.

Deja que la IA avance libremente cuando:

- El paso es pequeño y está bien definido (“agrega un botón”).
- Es un patrón común que la IA claramente ha visto mil veces.
- Un error es barato y evidente (lo verás en el momento en que ejecutes la aplicación).

Dirige manualmente —prompts más pequeños, revisión más cercana— cuando:

- El cambio toca datos que no puedes recuperar fácilmente (eliminar registros, migraciones).
- La IA sigue “arreglando” lo mismo en círculos.
- El comportamiento es sutil y no se notará con solo mirar la pantalla (cálculos de dinero, permisos, autenticación).
- Ya intentaste dos veces y las cosas están *empeorando*, no mejorando.

Una regla confiable: **cuanto mayor sea el radio de impacto, más pequeño debe ser el paso.**

Mantén la aplicación siempre funcionando

Esta es la disciplina que hace posible todo lo demás:

- **Ejecuta la aplicación después de cada paso.** No cada tres pasos. Cada uno.

- **Confirma solo estados funcionales.** Un commit es un punto de guardado al que puedes volver. Si un paso rompe algo que no puedes arreglar en uno o dos minutos, descártalo (`git restore .` o revierte) y vuelve a escribir el prompt con una petición más clara; no apiles un arreglo sobre una base rota.
- **Una sola preocupación por paso.** Si te da la tentación de “de paso arreglar esa otra cosa”, resiste. Termina la rebanada actual, confirma el cambio y luego empieza un paso nuevo.
- **Describe el error, no lo diagnostiques.** Cuando algo se rompe, dile a la IA exactamente lo que viste: “Después de hacer clic en la segunda estrella, el contador muestra 1, no 2.” Deja que ella encuentre la causa.

Confirmar cambios en incrementos pequeños

Los commits pequeños son tu red de seguridad. Cada uno debe ser un solo paso coherente:

```
git commit -m "Add star button to item cards"
git commit -m "Toggle favorite state on star click"
git commit -m "Add favorites counter to list header"
git commit -m "Persist favorites in localStorage"
```

Si el paso 4 sale mal, tus primeros tres pasos están a salvo y casi no pierdes nada. Los commits grandes y poco frecuentes son la razón por la que la gente termina teniéndole miedo a tocar su propio código.

Cuando una rebanada es demasiado grande

A veces eliges “un paso” y aun así sale mal: la IA reescribe media aplicación, el diff es ilegible y al ejecutarlo aparecen tres errores nuevos a la vez. Eso casi nunca es mala suerte. Es una señal de que la rebanada era demasiado grande.

La solución es cortarla más delgada. Si “agregar favoritos con persistencia” explota, eso no era una sola rebanada; era la función completa disfrazada de un solo prompt. Regresa a tu último commit bueno y divídelo:

```
# Don't ask for this in one shot:
"Add a favorites feature with a counter and localStorage persistence."

# Ask for this instead, one commit each:
"Add a star button to each item card. No behavior yet."
"Make the star toggle filled/outline with local state."
"Add a counter that reflects the filled stars."
"Persist the favorited IDs to localStorage and load them on startup."
```

Una buena regla general: si no puedes describir lo que hace un paso en una sola oración corta, son dos pasos. Y si el diff de un paso es demasiado largo para hojearlo en un minuto, el siguiente debería ser más pequeño. Dividir en rebanadas es una habilidad que mejora al sentir en dónde la última rebanada resultó demasiado gruesa.

La conclusión

Una función no es un solo prompt gigante: es una pila de rebanadas pequeñas y ejecutables. Divide, escribe el prompt para un paso, revisa, ejecuta, confirma, repite. Deja que la IA avance libremente en lo fácil y dirige con firmeza cerca de cualquier cosa riesgosa. Mantén la aplicación funcionando en cada punto de control, y siempre estarás a un pequeño “deshacer” de terreno firme.

En el próximo capítulo veremos qué hacer cuando un paso no sale como estaba planeado: depurar por vibe.

Repaso y práctica

Ideas clave

- Una función es una pila de rebanadas verticales pequeñas, no un solo prompt gigante: construye, ejecuta y *observa* funcionar cada rebanada antes de seguir adelante.
- Ejecuta el ciclo de construcción cada vez: dividir → elegir un paso → escribir el prompt → revisar el diff → ejecutar la aplicación → confirmar → repetir.
- Revisa los diffs como un revisor, no como un compilador: fíjate en archivos que no debería haber tocado, cambios de comportamiento silenciosos y un diff mucho más grande de lo pedido.
- Deja que la IA avance libremente en pasos pequeños, baratos y comunes; dirige manualmente (prompts más pequeños, revisión más cercana) donde sea que el radio de impacto sea grande.
- Confirma solo estados funcionales en incrementos pequeños: si una rebanada sale mal, descártala y vuelve a escribir el prompt en lugar de apilar un arreglo sobre una base rota.

Ponlo en práctica

Elige una función pequeña que quieras agregar a un proyecto y divídela en 3 a 5 pasos verticales, escribiendo cada uno como una sola oración corta. Construye *solo el primer paso*: escribe el prompt para él, ejecuta `git diff` y hojear el cambio, ejecuta la aplicación para confirmar que funciona y luego confirma el cambio con un mensaje de una línea. Detente ahí por ahora. El objetivo es sentir una sola vuelta limpia del ciclo: una rebanada ejecutable, un diff que realmente leíste, un commit al que puedes volver.

```
I want to add this feature: [describe the feature in one or two sentences].  
Don't write any code yet. First, slice it into 3 to 5 vertical steps, where  
each step is something I can build, run, and SEE working on its own. Order  
them so the app keeps working after every step. For each step give me a  
one-sentence description and name the first file you'd change. Then wait – I'll  
ask you to build them one at a time.
```

Depuración con IA



Todo programa que construyas se romperá. Eso no es una señal de que lo estés haciendo mal: es el estado normal del software. Los errores (bugs) no son un fallo de habilidad; son lo que ocurre cuando el código real se encuentra con datos reales, usuarios reales y casos límite que nadie imaginó. La diferencia entre una tarde frustrante y una solución de cinco minutos es el *método*. Cuando depuras haciendo *vibe coding*, no eres tú quien se queda mirando el stack trace hasta que tiene sentido —el stack trace es el informe de error que tu programa imprime cuando falla, algo que explicaremos más adelante—. Tú eres el director: reproduces el problema, reúnes la evidencia correcta y le entregas a la IA un caso bien enmarcado y concreto para que ella haga el trabajo pesado. Este capítulo te da un proceso repetible para exactamente eso, uno que funciona tanto si el bug es un error tipográfico de una línea como si es una condición de carrera enredada tres capas por debajo. (Una condición de carrera es un error de sincronización: dos cosas ocurren en un orden que no esperabas, como dos personas agarrando el último asiento a la vez.)

La mentalidad central

El error más grande que cometen los principiantes es pegar “no funciona, arrégalo” en el chat y esperar lo mejor. La IA no puede ver tu pantalla, tus datos, ni lo que esperabas que ocurriera. Solo sabe lo que le cuentas. Cuando no le das nada, hace lo único que puede: adivinar; y una suposición segura que resulta estar equivocada es más peligrosa que ninguna respuesta, porque tú actuarás según ella. Depurar con IA consiste sobre todo en **darle el contexto correcto** para que pueda razonar en lugar de adivinar.

Un buen reporte de bug —para un humano o para una IA— responde tres preguntas:

- **¿Qué hiciste?** (los pasos para provocarlo)
- **¿Qué esperabas que pasara?**
- **¿Qué pasó en realidad?** (el error, la salida incorrecta, el fallo)

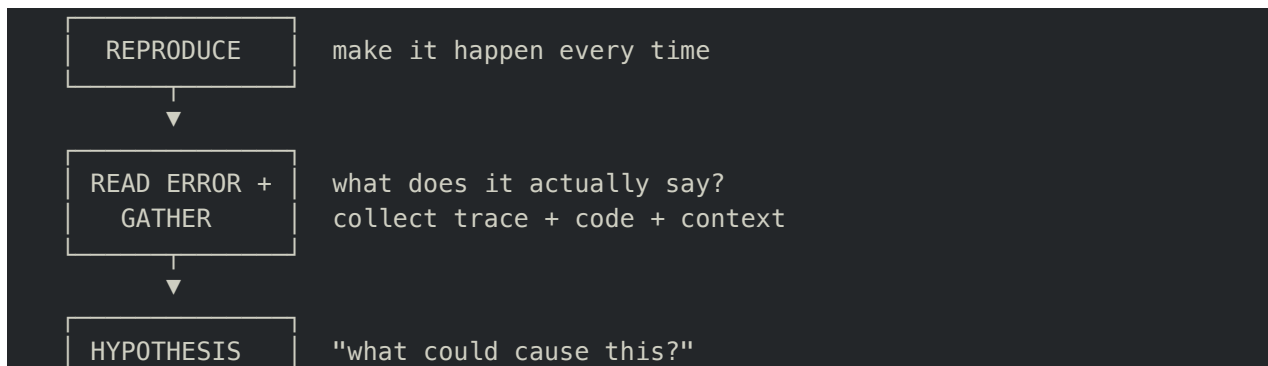
Si puedes responder esas tres preguntas, ya vas por delante de la mayoría de la gente. La razón por la que estas tres cosas importan es que un bug es, por definición, una *brecha* entre lo que esperabas y lo que ocurrió. La IA no puede encontrar una brecha si no puede ver ambos lados. “Está roto” describe solo un lado; “esperaba una pantalla de bienvenida, obtuve una página en blanco y este error” describe ambos lados, y la brecha prácticamente se nombra sola.

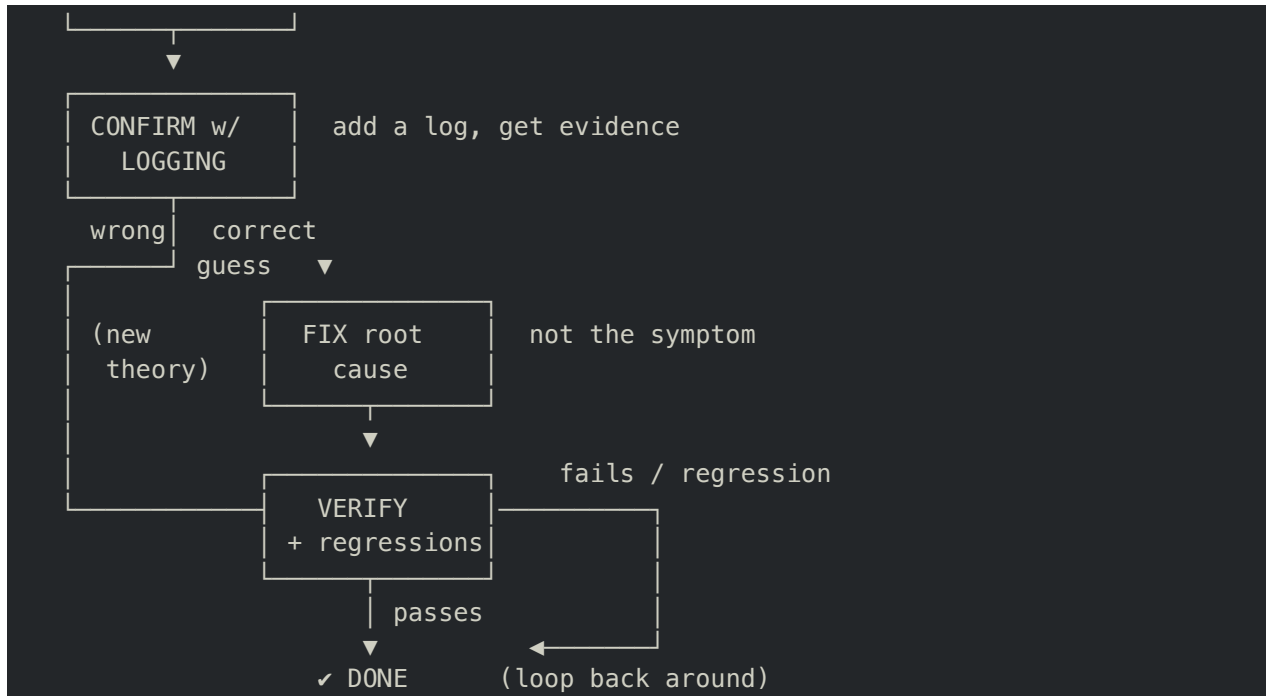
Una lista de verificación para depurar

Recorre estos pasos en orden. Saltar directamente a “arréglalo” es lo que hace que la IA dé vueltas en círculo.

1. **Reprodúcelo de forma confiable.** Encuentra los pasos exactos que provocan el bug cada vez. Un bug que no puedes reproducir es uno que no puedes confirmar que arreglaste; solo estarás adivinando si desapareció. Si ocurre de forma intermitente, busca la variable que lo activa: una entrada específica, un estado de sesión iniciada versus cerrada, una red lenta.
2. **Lee el error real.** No lo hojees. El mensaje de error y la *parte superior* del stack trace suelen nombrar el archivo y la línea donde algo salió mal. Los principiantes a menudo se saltan la frase más útil de la pantalla porque parece intimidante.
3. **Reúne el contexto.** Recopila el texto del error, el stack trace, el código relevante y una descripción de una línea de lo que esperabas.
4. **Dale a la IA el panorama completo.** Pega todo lo anterior en un solo mensaje. Deja que forme una hipótesis.
5. **Formula una hipótesis, no un arreglo.** Pregunta “¿qué podría estar causando esto?” antes de “arregla esto”. Un arreglo equivocado sobre una teoría equivocada desperdicia tiempo, y a veces añade un *segundo* bug encima del primero.
6. **Añade registros (logging) para probar la hipótesis.** Confirma qué está haciendo realmente el código antes de cambiarlo.
7. **Aplica búsqueda binaria al problema.** Acota *dónde* se rompe cortando repetidamente por la mitad el área sospechosa.
8. **Arregla la causa raíz, no el síntoma.** Asegúrate de que el arreglo aborde *por qué* se rompió, no solo el fallo visible.
9. **Verifica el arreglo y revisa si hay regresiones.** Vuelve a ejecutar los pasos de reproducción. Luego asegúrate de que las funciones cercanas sigan funcionando; una *regresión* es cuando un arreglo rompe silenciosamente algo que solía funcionar, como parchar una fuga y reventar una tubería justo al lado.

Esa lista de verificación es en realidad un solo ciclo que recorres hasta que el bug desaparece. Cada casilla le pasa evidencia a la siguiente, y una verificación fallida simplemente te devuelve al ciclo con lo que aprendiste:





El orden no es arbitrario. Cada paso produce la evidencia que necesita el siguiente. Reproducir te da algo que leer; leer te da contexto que reunir; el contexto le da a la IA algo sobre lo que formular una hipótesis. Sáltate un paso y estarás alimentando al siguiente con un vacío.

Alimentar a la IA con el contexto correcto

Aquí es donde reside la mayor parte de tu ventaja. Cuando reportes un bug, incluye:

- **El mensaje de error, textual.** Cópialo exactamente, no lo parafrasees. Parafrasear un error es como describir una foto por teléfono: te dejarás fuera el único detalle que importaba.
- **El stack trace**, especialmente las primeras líneas.
- **El código relevante:** la función que lanzó el error, más cualquier cosa que esta llame.
- **Lo que esperabas frente a lo que pasó.**

Así es como se ve realmente un buen prompt de depuración:

I'm getting an error when I submit the signup form. Here's the full output:

```

TypeError: Cannot read properties of undefined (reading 'email')
    at validateUser (src/auth/validate.js:14:23)
    at handleSignup (src/routes/signup.js:31:10)
    at processTicksAndRejections (node:internal/process/task_queues:95:5)
  
```

Steps to reproduce: fill in the form, click "Sign up".

Expected: a "Welcome" message.

Actual: the page goes blank and the error above appears in the console.

Here is validate.js (lines around 14):

[paste the function]

And here is the handleSignup function that calls it:

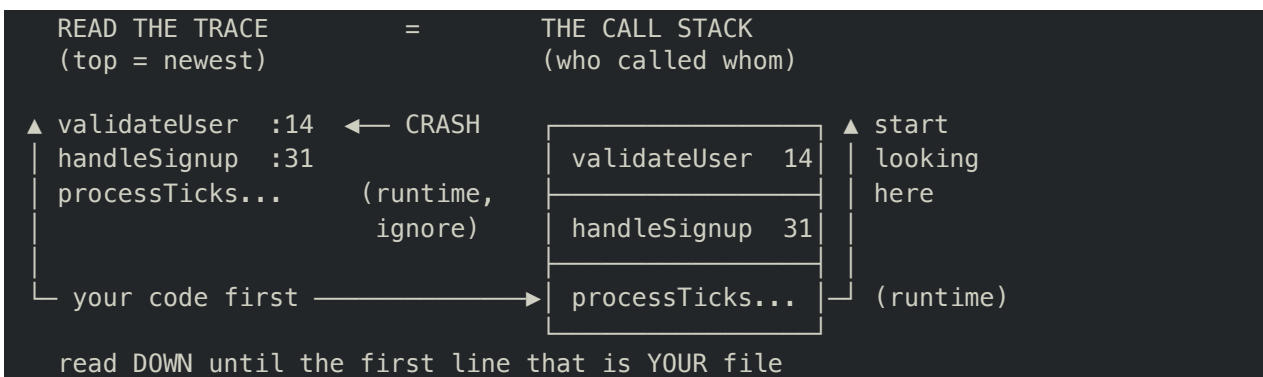
[paste the function]

What could be causing this, and how do I confirm it before changing anything?

Fíjate en esa última línea. Estás pidiendo una *hipótesis* y un *paso de confirmación*, no una edición inmediata. Eso mantiene a la IA honesta.

Ayuda saber cómo *leer* el trace que estás pegando. Un stack trace es un historial de llamadas, del más reciente al más antiguo: la línea de arriba es donde el programa realmente explotó, y cada línea debajo es la función que llamó a la de arriba. Así que en el ejemplo, `validateUser` es donde se leyó `email` de algo indefinido, y `handleSignup` es quien llamó a `validateUser`. Los cuadros de abajo de tu propio código —como `processTicksAndRejections`— pertenecen al entorno de ejecución del lenguaje; normalmente puedes ignorarlos. Lee de arriba hacia abajo hasta llegar a la primera línea que apunte a *tu* archivo, y ahí es casi siempre donde debes empezar a buscar.

El mismo trace, leído como una pila de llamadas, hace que el orden sea obvio. El fallo está arriba; cada cuadro debajo es quien llamó al de arriba:



Leer errores que no entiendes

Tarde o temprano te encontrarás con un error que parece sopa de letras: un nombre de tipo largo, un stack lleno de internos del framework, un mensaje en una jerga que nunca has visto. No dejes que eso te detenga. El error es información, no un veredicto sobre tu competencia. Cuando el mensaje mismo sea opaco, pégalo y pídele a la IA que lo *traduzca antes de arreglarlo*:

I don't understand this error. Before suggesting any fix, explain in plain language: what is it actually saying, which line is it pointing at, and what kinds of things typically cause this exact error?

[paste the full error and stack trace]

Pedir primero el significado en lenguaje llano hace dos cosas. Te enseña el patrón, de modo que la próxima vez que veas ese error lo reconocerás tú mismo —y a lo largo de unos meses así es como realmente aprendes a leer tu propio stack—. Y obliga a la IA a fundamentar su arreglo en lo que el error *dice*, en lugar de saltar a un parche genérico. Cuidado con la trampa de error más común: un error que apunta a la línea 14 a menudo fue *causado* en un lugar completamente distinto. La línea en el trace es donde el programa notó el problema, no siempre donde el problema comenzó.

Hipótesis, registros y búsqueda binaria

Una vez que la IA propone una causa, no la aceptes sin más. **Pruébala de forma barata.** La prueba más barata suele ser una línea de registro (log).

Pídele a la IA que añada registros temporales:

“Antes de cambiar nada, añade un `console.log` (o `print`) que muestre el valor de `user` justo antes de la línea 14, para que podamos ver si en realidad es indefinido.”

Ejecútalo, pega el resultado de vuelta. Ahora tienes *evidencia*, no una teoría. Quizás `user` es indefinido, confirmando la hipótesis. Quizás está bien y el verdadero culpable está un nivel más arriba. De cualquier forma, has avanzado. Registra con generosidad mientras investigas: el valor de la variable, y una etiqueta que diga cuál es y dónde está el registro. Un `console.log(user)` desnudo que imprime `undefined` es mucho menos útil que `console.log("user at validate.js:14 =", user)`. Y recuerda retirar los registros temporales una vez que el bug esté arreglado; el ruido de depuración sobrante es su propio pequeño desorden.

Cuando el bug vive en algún lugar de un proceso largo y no puedes saber dónde, usa la **búsqueda binaria**. Corta la región sospechosa por la mitad: añade un registro en el punto medio.

La idea: cada verificación descarta la mitad de lo que queda, así que un espacio de búsqueda enorme colapsa rápido. Aquí un bug se esconde en algún lugar entre el paso 1 y el paso 8; tres verificaciones lo acorralan:

```
step:  1  2  3  4  5  6  7  8      a bug is somewhere in here
      [XXXXXXXXXXXXXXXXXXXXX]

check 1 — log at 4 — OK →  bug is AFTER 4, drop 1-4
                        [XXXXX] (5 6 7 8 left)

check 2 — log at 6 — BAD →  bug is AT/BEFORE 6, drop 7-8
                        [XXXX]  (5 6 left)

check 3 — log at 5 — OK →  bug is at step 6  ✓ found

8 steps → 3 checks.  1000 steps → about 10 checks.
```

¿El valor se veía correcto ahí? Entonces el bug está *después*; busca en esa mitad. ¿Se veía mal ya? El bug está *antes*; busca en esa mitad. Cada paso reduce a la mitad el espacio de búsqueda, así que incluso una ruta de mil líneas se acota en unas diez verificaciones. Dile a la IA explícitamente: “Ayúdame a bisecar esto, ¿cuál es el punto medio donde debería registrar?” La misma técnica funciona en el historial, no solo en el código: si una función funcionaba la semana pasada y se rompió hoy, bisecciona tus commits para encontrar el cambio exacto que introdujo el bug.

Causa raíz frente a síntoma

La trampa más tentadora al depurar es el parche que hace que el error *desaparezca* sin arreglar la causa. Si `user` es indefinido y lo “arreglas” con `if (user) { ... }`, el error se detiene, pero ahora los registros de usuario no hacen nada en silencio, lo cual es peor: un fallo ruidoso al menos te decía que algo estaba mal, y ahora nada te avisa. La pregunta real es *¿por qué user es indefinido aquí?* Quizás la consulta a la base de datos falló. Quizás un campo fue renombrado. Quizás falta un `await` más arriba en la cadena.

Cuando la IA propone un arreglo, pregúntale:

- “¿Esto arregla la causa o esconde el síntoma?”
- “¿Cuál fue la razón *original* por la que el valor estaba mal?”
- “¿Podría esta misma causa raíz romper alguna otra cosa?”

Un arreglo de causa raíz suele hacer que el código sea *más simple* o *más claro*. Un parche de síntoma suele añadir una guardia, un `try/catch` que traga errores, o un valor por defecto que tapa el problema real. Sospecha de este segundo tipo. Hay una señal que vale la pena aprender: si un arreglo añade código cuyo único trabajo es *evitar* un fallo en lugar de *hacer lo correcto*, probablemente estás parchando un síntoma. Los arreglos reales tienden a eliminar por completo el estado imposible, no solo a sobrevivirlo.

Cuando la IA da vueltas en círculo

A veces la IA sugiere un arreglo, no funciona, sugiere otro, tampoco funciona, y ya llevas tres ciclos de “prueba esto en su lugar”. Esto es una señal, no un callejón sin salida. Haz esto:

- **Detente y reinicia el contexto.** Empieza un mensaje nuevo resumiendo todo lo que ahora *sabes* con certeza (confirmado por los registros), no todo lo que has intentado. Un chat largo lleno de intentos fallidos envenena activamente la siguiente respuesta de la IA; sigue haciendo coincidir patrones con los callejones sin salida de su propio historial.
- **Declara lo que has descartado.** “Confirmamos que user es indefinido y que la consulta a la BD devuelve null. La cadena de la consulta se ve bien. Así que el problema está antes de la consulta.” Esto evita que la IA vuelva a sugerir callejones sin salida.
- **Pídele que explique, no que arregle.** “Explícame, paso a paso, cómo user obtiene su valor desde la solicitud hasta la línea 14.” Forzar una explicación a menudo hace visible la brecha.
- **Añade más registros, no más suposiciones.** Cuando las teorías se agotan, la evidencia es el camino a seguir.
- **Reduce el problema.** Pídele a la IA que te ayude a construir el fragmento más pequeño posible que aún reproduzca el bug. A menudo el bug se vuelve obvio una vez que desaparece el ruido, y si no se reproduce en la versión pequeña, has aprendido que la causa está en lo que quitaste.

Entender el arreglo antes de conservarlo

Un arreglo que funciona pero que no entiendes es un préstamo, no una solución, y el interés vence la próxima vez que algo cercano se rompa. Como la IA hace la escritura, es fácil aceptar una marca de verificación en verde y seguir adelante sin saber nunca *por qué* el código ahora está en verde. Resiste esa tentación. Antes de conservar un arreglo, haz que la IA lo justifique:

- **“Explica por qué esto lo arregla, en una o dos frases.”** Si la explicación es vaga —“esto lo hace más robusto”, “esto maneja el caso límite”— insiste. Una explicación real nombra la causa específica y cómo el cambio la elimina.
- **“¿Cuál fue el bug, en términos de causa y efecto?”** Deberías poder repetir la respuesta con tus propias palabras. Si no puedes, aún no entiendes tu propio programa.
- **“¿Qué más toca esta ruta de código que debería volver a probar?”** Esto convierte un arreglo en una verificación de regresión.

Esto es la filosofía de “entiende lo que envías” aplicada a su caso más difícil. No necesitas ser capaz de *escribir* el arreglo desde cero, ese es el trabajo de la IA. Pero sí necesitas tener en la cabeza la historia de causa y efecto, porque tú eres quien estará ahí a las 2 a.m. cuando vuelva a romperse, y lo único que ayuda entonces es saber cómo funciona realmente la cosa.

La conclusión

Depurar con IA no se trata de saber la respuesta. Se trata de ejecutar un proceso disciplinado: reproducir, reunir contexto, formular una hipótesis, confirmar con registros, bisecar y arreglar la causa raíz; y luego entender el arreglo lo suficientemente bien como para conservarlo con honestidad. Tu trabajo como director es mantener la investigación anclada en evidencia y reconocer cuándo el ciclo se ha estancado. Haz eso, y la mayoría de los bugs dejarán de ser misterios y pasarán a ser solo otro paso en la construcción de software real.

Resumen y práctica

Puntos clave

- Un bug es una brecha entre lo que esperabas y lo que pasó; dale a la IA ambos lados más los pasos exactos, nunca solo “está roto”.

- Pega el error textualmente y la parte superior del stack trace; la primera línea que apunta a *tu* archivo es casi siempre donde empezar.
- Confirma antes de cambiar: añade una línea de registro etiquetada para probar una hipótesis, y bisecciona (en el código o en el historial de commits) para reducir a la mitad el espacio de búsqueda en cada paso.
- Arregla la causa raíz, no el síntoma: una guardia o un try/catch que solo detiene un fallo suele tapar el problema real.
- Cuando la IA da vueltas en círculo, reinicia el contexto con lo que *sabes* con certeza, pídele que explique en lugar de arreglar, y añade evidencia en lugar de suposiciones.

Ponlo en práctica

La próxima vez que algo se rompa, escribe el reporte del bug *antes* de pedir un arreglo. En un chat nuevo, dale a la IA tres cosas: los pasos para reproducirlo, lo que esperabas y lo que pasó en realidad, con el error copiado textualmente y los primeros cuadros del stack incluidos. Luego, en lugar de pedir “arreglalo”, pide una hipótesis y una forma barata de confirmarla (normalmente una línea de registro). Ejecuta esa verificación, pega el resultado de vuelta, y solo entonces deja que la IA proponga un arreglo. Fíjate en cuánto más directo es el camino comparado con “no funciona, ayuda”.

```
I have a bug. Before suggesting any fix, give me a hypothesis and a cheap way
to confirm it (such as a log line) – don't edit code yet.

Steps to reproduce: [what you did]
Expected: [what should have happened]
Actual: [what happened]

Full error and stack trace:
[paste verbatim]

Relevant code:
[paste the function that failed, plus anything it calls]

What could be causing this, and what should I log to confirm it before we change anything?
```

Infraestructura de servidores



Hiciste vibe coding de algo que funciona en tu laptop. Ahora quieres que otras personas lo usen. Esa brecha —entre “funciona en mi máquina” y “funciona en internet”— es de lo que trata la infraestructura. La buena noticia: no necesitas una carrera en DevOps para cruzarla. Necesitas un modelo mental que funcione y la disposición a dejar que la IA se encargue de la configuración engorrosa.

Este capítulo desmitifica el panorama del hosting, te da las ventajas y desventajas en lenguaje simple, y te entrega una ruta por defecto que lleva a un constructor solitario del prototipo a producción sin ahogarse en opciones.

Los cinco lugares donde puede vivir el código

Casi todos los productos de hosting en el mundo son una variación de una de estas cinco categorías. Aprende esto y las páginas de marketing dejan de ser confusas.

- **Hosting estático** — Archivos planos (HTML (HyperText Markup Language, la estructura de la página), CSS (Cascading Style Sheets, el estilo), JavaScript, imágenes) que se sirven directo al navegador. Ningún servidor “ejecuta” tu código; solo entrega archivos. Piénsalo como una carpeta en internet. Como no hay nada que pueda fallar ni nada que parchear, esto es lo más confiable que puedes poner en línea. Ejemplos: GitHub Pages, Cloudflare Pages, Netlify, Vercel (modo estático).
- **Funciones edge** — Fragmentos diminutos de código que corren en servidores físicamente cercanos a tu usuario, repartidos por todo el mundo. Arrancan casi instantáneamente y son excelentes para lógica ligera: verificaciones de autenticación, redirecciones, pequeñas llamadas a API (Application Programming Interface — la puerta por la que un programa habla con otro). La trampa: corren en un

entorno reducido, así que no todas las librerías funcionan ahí. Ejemplos: Cloudflare Workers, Vercel Edge Functions, Deno Deploy.

- **Funciones serverless** — Código que se ejecuta bajo demanda y desaparece cuando está inactivo. Te cobran por solicitud, no por hora. Tú escribes una función; la plataforma se encarga de las máquinas. Genial hasta que una sola solicitud necesita correr durante minutos: la mayoría de las plataformas cortan la función tras un tiempo límite (a menudo de 10 a 60 segundos). Ejemplos: AWS Lambda, Google Cloud Functions, Vercel Functions.
- **Contenedores** — Tu app más todo su entorno (la versión correcta del lenguaje, librerías, herramientas del sistema) empaquetados en una caja portátil llamada contenedor. La plataforma mantiene esa caja funcionando. Más control, un poco más de configuración. El archivo que define la caja es un Dockerfile, y la IA los escribe bien. Ejemplos: Fly.io, Railway, Render, Google Cloud Run.
- **Máquinas virtuales (VM)** — Una computadora completa que alquilas en un centro de datos. Instalas y gestionas todo tú mismo: el sistema operativo, las actualizaciones de seguridad, el servidor web, todo. Máximo control, máxima responsabilidad. Ejemplos: AWS EC2, DigitalOcean Droplets, Hetzner.

Puestas una junto a otra, las cinco categorías intercambian *menos que gestionar por más control* a medida que bajas:

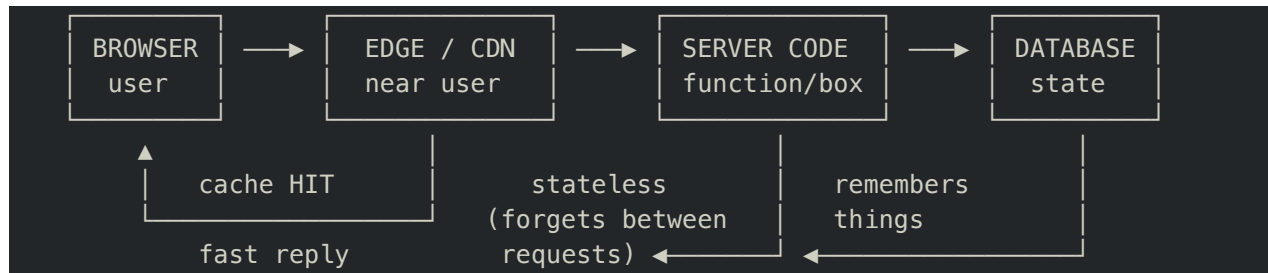


Una regla general aproximada para el costo: mientras más gestiona la plataforma por ti, más barato es cuando nadie usa tu app, y más se encarga ella de preocuparse cuando todos la están usando.

Un par de términos de jerga, descifrados

- **Cold start (arranque en frío)** — Cuando el código no ha corrido en un rato, la plataforma tiene que despertarlo. Esa primera solicitud es más lenta (de una fracción de segundo a unos pocos segundos). Las funciones edge casi no sufren esto; las funciones serverless a veces sí; los contenedores y VMs siempre activos no lo sufren. Para la mayoría de las apps esto es invisible; para un flujo de pago que debe sentirse ágil, puede importar.
- **Escalado** — Manejar más usuarios sin caerse. El escalado *automático* significa que la plataforma añade capacidad por ti. El escalado *manual* significa que tú haces clic en botones o simplemente se rompe bajo carga. “Escala a cero” es una frase relacionada que verás mucho: cuando nadie usa tu app, no cuesta nada.
- **Estado** — Datos que necesitan persistir (una base de datos, archivos subidos, una sesión de usuario). Las capas estáticas y serverless suelen ser “sin estado” (stateless): olvidan todo entre solicitudes, así que el estado vive en una base de datos o servicio de almacenamiento separado. Por eso casi toda app real es “tu hosting más una base de datos en algún lugar”, no una sola caja mágica.
- **Región** — En qué parte del mundo se encuentran físicamente tu código y tus datos. Afecta la velocidad (más cerca es más rápido) y a veces la ley (algunos datos deben permanecer en un país específico). Para un primer proyecto, elige la región más cercana a tus usuarios y sigue adelante.

Aquí está todo el recorrido de una solicitud, y dónde vive cada pieza. El navegador habla con un edge cercano, que sirve archivos en caché o despierta tu código, el cual acude a la base de datos por cualquier cosa que deba recordar:



La tabla de compensaciones

Tipo de hosting	Costo (inactivo □ ocupado)	Escalado	Cold starts	Complejidad	Mejor para
Estático	Gratis □ muy barato	Automático, sin esfuerzo	Ninguno	La más baja	Sitios de marketing, documentación, blogs, frontends
Funciones edge	Nivel gratuito □ barato	Automático, global	Insignificante	Baja	Autenticación, redirecciones, APIs ligeras, personalización
Serverless	\$0 en inactividad □ escala con el uso	Automático	A veces notable	Baja-media	APIs, webhooks, tráfico intermitente/impredecible
Contenedores	Pagas mientras corre	Automático-ish, configurable	Ninguno (si se mantiene caliente)	Media	Backends completos, tareas de larga duración, entornos de ejecución personalizados
VMs	Pagas 24/7	Manual	Ninguno	La más alta	Hardware especial, software heredado, control total

El patrón: a medida que bajas por la tabla, cambias *simplicidad y bajo costo en inactividad* por *control y flexibilidad*. La mayoría de los *vibe coders* debería esforzarse por quedarse cerca de la parte superior.

Cuándo elegir cuál

- **¿Solo un sitio web o frontend?** Hosting estático. Listo. Es gratis, rápido y casi imposible de romper.
- **¿Necesitas un poco de lógica de backend: un formulario de contacto, una clave de API mantenida en secreto, una verificación de autenticación?** Funciones edge o serverless. Te mantienes en un nivel gratuito generoso y escalas a cero cuando nadie lo usa.

- **¿Una app real con base de datos, tareas en segundo plano, o un framework que espera un servidor de ejecución prolongada?** Una plataforma de contenedores como Railway, Render o Fly.io. Este es el punto óptimo para la mayoría de los proyectos de “producto real”.
- **¿Necesitas una GPU específica, software de sistema especial, o control total?** Una VM. Solo ve aquí cuando algo de lo anterior genuinamente no puede hacer el trabajo, porque ahora *tú* eres el administrador del sistema.

Una regla útil: **elige la opción más simple que pueda hacer el trabajo, y solo baja de nivel cuando choques con un muro real**, no uno imaginado.

¿Cómo se ve realmente un “muro real”? Algunos ejemplos honestos: tu build produce un servidor que necesita permanecer corriendo (un backend de chat que mantiene conexiones en vivo), así que lo estático no sirve. O una función sigue chocando con su tiempo límite porque una solicitud realiza trabajo real (redimensionar un video, generar un PDF), así que te gradúas a un contenedor. O el límite de solicitudes de tu nivel gratuito está genuinamente agotado por tráfico real. Nota lo que estos casos tienen en común: cada uno es una medición, no una corazonada. “Podría crecer algún día” no es un muro.

Cómo la IA cablea los despliegues y la configuración

Aquí es donde brilla el vibe coding. La configuración de despliegue es exactamente el tipo de trabajo preciso, bien documentado y lleno de repeticiones en el que la IA sobresale. Tú diriges; ella escribe.

Las instrucciones (prompts) efectivas suenan así:

- “Agrega un Dockerfile para esta app de Node y una configuración de Railway para desplegarla.”
- “Escribe un workflow de GitHub Actions que despliegue la carpeta dist/ a Cloudflare Pages en cada push a main.”
- “Estoy recibiendo un timeout de cold-start en mi función serverless; ¿qué es probable que esté mal y cómo arreglo la configuración?”

Aquí tienes el tipo de archivo que la IA producirá para ti: un workflow de despliegue que publica un sitio estático automáticamente cada vez que subes código:

```
# .github/workflows/deploy.yml
name: Deploy
on:
  push:
    branches: [main]
jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 20
      - run: npm ci && npm run build
      - name: Publish to Cloudflare Pages
        uses: cloudflare/wrangler-action@v3
        with:
          apiToken: ${ secrets.CF_API_TOKEN }
          command: pages deploy dist --project-name=my-app
```

No necesitas memorizar esta sintaxis. Necesitas reconocer qué hace: en cada push a main, descarga tu código, lo construye y publica el resultado. Cuando algo se rompe, pegas el error de vuelta a la IA y le pides que arregle la configuración.

Muchos constructores se saltan por completo el archivo de GitHub Actions al principio y simplemente despliegan desde su propia terminal con un solo comando. A menudo esa es la forma más rápida de ver algo en vivo el primer día. El mismo despliegue de Cloudflare Pages se ve así desde la línea de comandos:

```
# install the CLI once
npm install -g wrangler

# build your site, then push the output folder live
npm run build
wrangler pages deploy dist --project-name=my-app
```

La primera ejecución te pedirá iniciar sesión a través de tu navegador; después de eso es un solo comando. La compensación es sencilla: el comando de terminal es genial para “públicalo ahora”, y el archivo de GitHub Actions es genial para “públicalo automáticamente para siempre”. La mayoría de los proyectos empiezan con lo primero y añaden lo segundo una vez que los despliegues se vuelven rutinarios. Pídele a la IA lo que se ajuste a dónde estás.

Algunos consejos de dirección que ahorran dolores de cabeza reales:

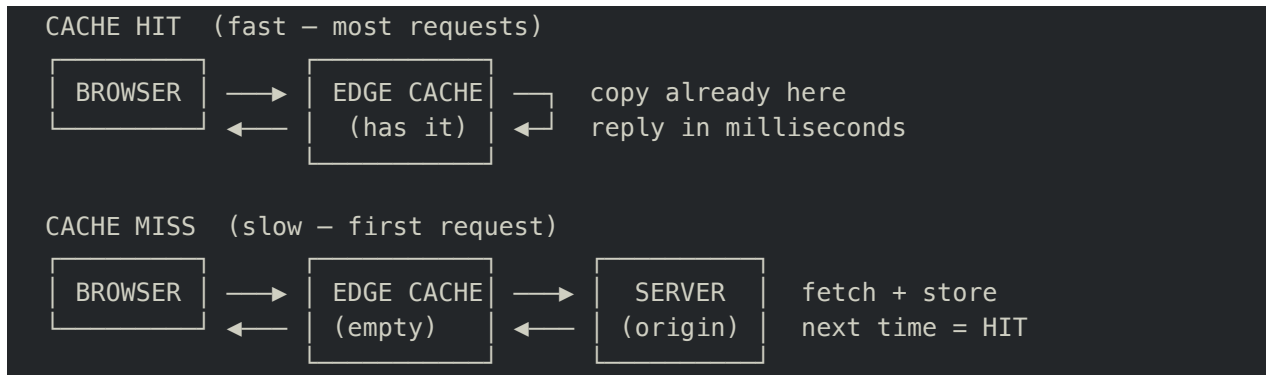
- **Haz que la IA explique antes de actuar.** “Antes de escribir la configuración, dime qué tipo de hosting le queda a este proyecto y por qué.” Esto detecta el sobre-diseño temprano.
- **Mantén los secretos fuera del código.** Las claves de API y contraseñas van en las variables de entorno / almacén de secretos de la plataforma, nunca en archivos que subes a control de versiones. Pídele a la IA que use una referencia a un secreto, como el CF_API_TOKEN de arriba.
- **Despliega temprano, despliega seguido.** Ten un “hola mundo” en vivo el primer día. Un pipeline que ha funcionado desde el inicio es mucho más fácil de depurar que uno ensamblado en el lanzamiento.
- **Pide de antemano la historia de rollback.** “Si este despliegue está roto, ¿cómo vuelvo a la versión que funcionaba?” Las buenas plataformas mantienen tus despliegues anteriores a un clic de distancia. Saber que existe el botón de deshacer antes de necesitarlo convierte un pánico de las 2 a.m. en un simple encogimiento de hombros.

Una ruta por defecto recomendada para el constructor solitario

Aquí hay una ruta que ha funcionado para incontables proyectos de una sola persona. Cada paso se toma solo cuando realmente lo necesitas.

1. **Prototipo: estático + un servicio de backend gestionado.** Pon tu frontend en hosting estático (Cloudflare Pages, Vercel o Netlify). Para datos y autenticación, apóyate en un backend gestionado como Supabase o Firebase para evitar correr tú mismo una base de datos. Costo: prácticamente cero.
2. **Producción temprana: agrega funciones edge o serverless.** Cuando necesites lógica del lado del servidor —llamadas secretas a APIs, webhooks, autenticación personalizada—, agrega funciones en la misma plataforma. Sigue escalando a cero, sigue siendo barato, sigue siendo un solo despliegue.
3. **Producto real: mueve el backend a una plataforma de contenedores.** Cuando la app supere a las funciones (procesos de larga duración, un framework que quiere un servidor persistente, tareas en segundo plano), contenerízala y despliégala en Railway, Render o Fly.io. Haz que la IA escriba el Dockerfile y la configuración de la plataforma.
4. **Escala o especializa: solo entonces considera VMs o un proveedor de nube como AWS.** Aquí es donde salta la complejidad (y tu factura). La mayoría de los constructores solitarios nunca necesitan este paso, y eso es señal de estar haciendo bien las cosas, no una limitación.

Una CDN (Content Delivery Network, red de distribución de contenido) se sitúa frente a esta ruta y cambia la velocidad drásticamente según si ya tiene una copia. Un acierto de caché (cache HIT) responde instantáneamente desde el edge; un fallo de caché (cache MISS) tiene que viajar todo el camino de vuelta hasta tu servidor primero:



El error que hay que evitar es empezar en el paso 4 porque supuestamente es lo que usan los “ingenieros de verdad”. Lo cierto es lo contrario: empezar simple es la jugada de un experto. Siempre puedes graduarte a más control, pero no puedes recuperar fácilmente los meses que perderías luchando contra una infraestructura que no necesitabas.

Qué significa realmente “gestionado” (y qué sigue siendo tuyo)

Un servicio gestionado es uno donde la plataforma se encarga por ti de las partes aburridas y peligrosas: los servidores, los parches de seguridad, el escalado, los respaldos. “Serverless”, “base de datos gestionada” y “plataforma como servicio” son todos sabores del mismo trato: cedes las operaciones y sigues construyendo. Para un constructor solitario esto no es pereza; es apalancamiento. Las horas que no pasas aplicando actualizaciones de seguridad de Linux son horas que dedicas al producto real.

Pero “gestionado” no es “mágico”. Unas cuantas cosas siguen siendo tu responsabilidad sin importar cuánto gestione la plataforma:

- **Tus datos.** La plataforma mantiene la base de datos funcionando; asegurarte de tener un respaldo que realmente hayas probado a restaurar es tarea tuya. Haz clic en el botón de “exportar” una vez y confirma que el archivo no está vacío.
- **Tus secretos.** Las claves de API viven en el almacén de secretos de la plataforma, pero rotar una que se filtre, y nunca pegar una en un repositorio público o una captura de pantalla, es tu trabajo.
- **Tu factura.** El auto-escalado es maravilloso hasta que auto-escala tu factura. Configura alertas y topes de gasto: la siguiente sección trata enteramente de esto, porque es el fallo que más duele.
- **Tu comprensión.** Cuando algo se rompe a medianoche, “la plataforma se encarga” solo llega hasta cierto punto. No necesitas ser un experto, pero deberías poder leer los registros (logs) y describirle el fallo a la IA con claridad. Esa es toda la filosofía de “entiende lo que lanzas” resumida en una frase.

La mentalidad saludable: externaliza las *operaciones*, no la *comprensión*. Deja que la plataforma corra las máquinas; tú sigues siendo la persona que sabe qué están haciendo las máquinas.

Una lista de verificación rápida antes del lanzamiento

Antes de decirle a la gente que visite tu URL en vivo, recorre esta lista. Nada de esto requiere experiencia en DevOps; la mayoría es un solo prompt a la IA o un clic en un panel de control.

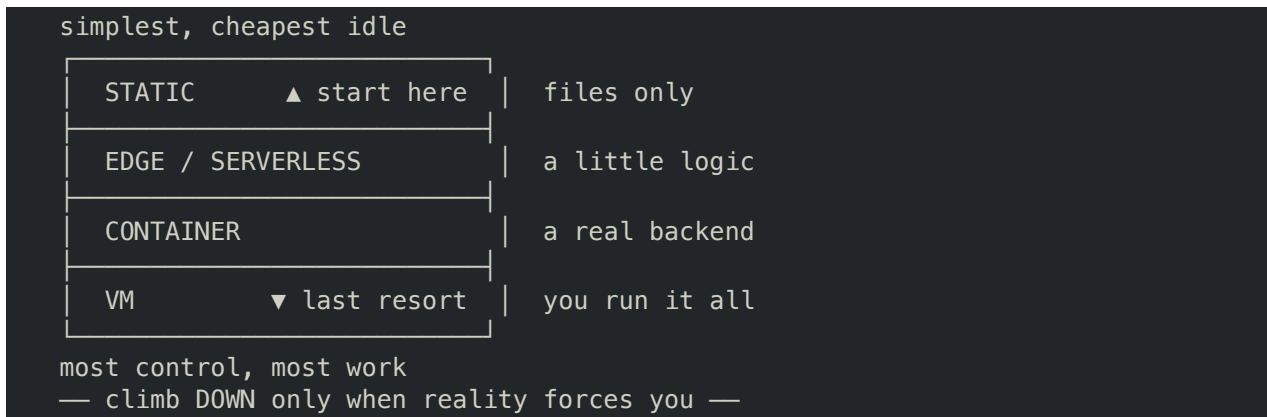
- **Realmente carga.** Abre la URL real (Uniform Resource Locator, la dirección web) en un navegador nuevo (o tu teléfono), no solo tu vista previa local. La cantidad de lanzamientos “funciona en mi máquina” que dan error 404 en público es legendaria.
- **Los secretos no están en el código.** Busca en tu repositorio tus propias claves antes de subir el código. Pídele a la IA: “escanea este proyecto en busca de claves de API, contraseñas o tokens escritos directamente en el código.”
- **Los topes de gasto están configurados.** Un tope duro en cada API de pago y una alerta de facturación en tu cuenta de hosting. Primero el tope, luego la celebración.

- **HTTPS está activado.** HTTPS (HyperText Transfer Protocol Secure) es la versión cifrada del sistema básico de entrega de la web; es lo que pone el candado en la barra de direcciones. Las plataformas modernas hacen esto automáticamente, pero confirma que el pequeño ícono de candado esté ahí. Los navegadores castigan a los sitios que no están cifrados.
- **Sabes cómo hacer rollback.** Encuentra ahora el botón de despliegue anterior, mientras nada está en llamas.
- **Hay una forma de que te contacten.** Una herramienta de registro o reporte de errores, incluso una básica, para que la primera señal de un problema no sea un usuario enojado. Pídele a la IA que conecte el registro de errores para tu stack.

Esta no es una auditoría de producción exhaustiva; es la lista corta que atrapa los errores vergonzosos, costosos y fácilmente evitables. Ejecútala en cada lanzamiento hasta que se vuelva un reflejo.

La conclusión

La infraestructura es una escalera, no un salto. Lo estático arriba, las VMs abajo, y la mayor parte de tu trabajo viviendo felizmente en los peldaños superiores.



Conoce las cinco categorías, elige la más simple que haga el trabajo, deja que la IA escriba la configuración, y lanza algo en vivo tan pronto como puedas. El resto es solo bajar un peldaño cuando —y solo cuando— la realidad te obligue a hacerlo.

Repaso y práctica

Puntos clave

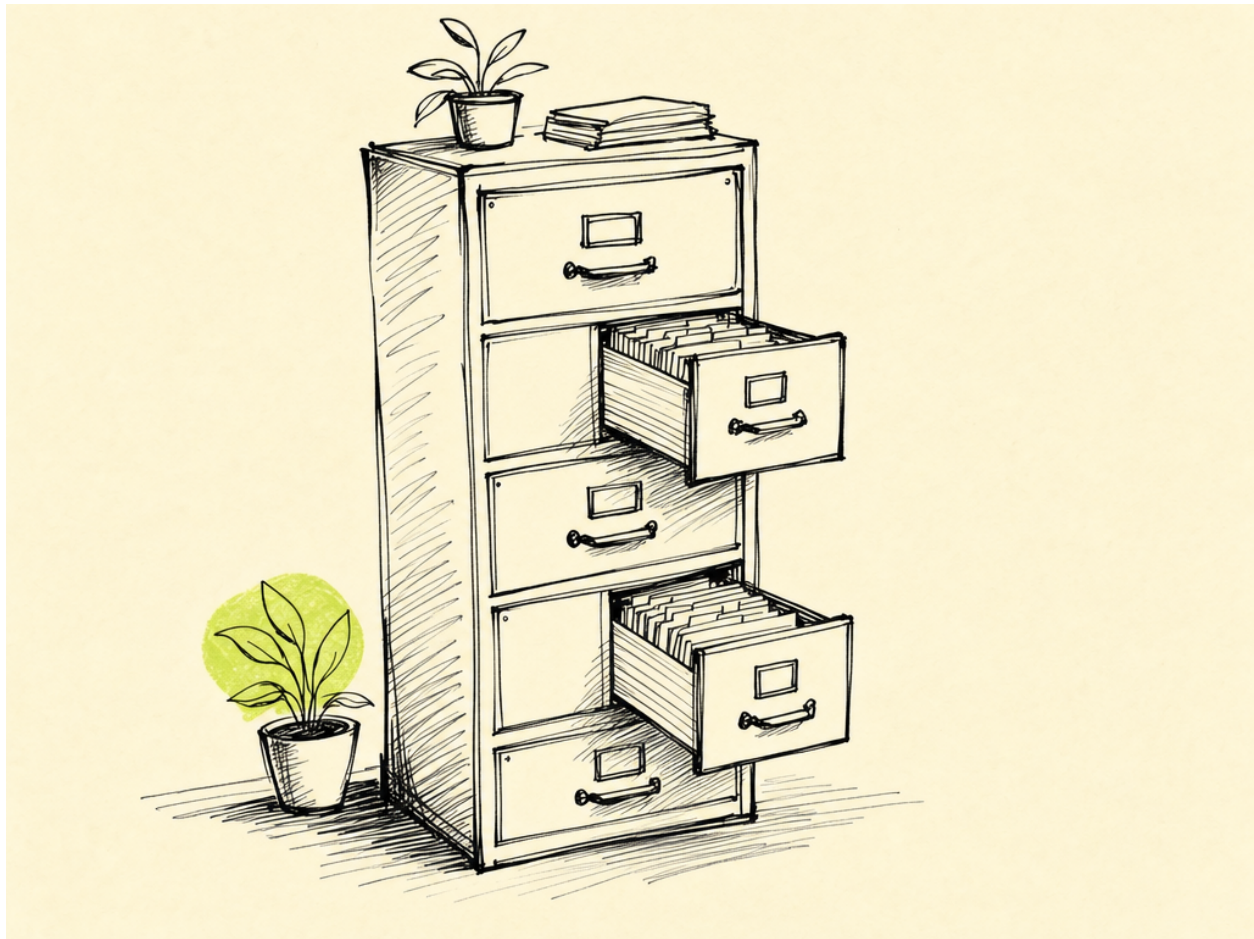
- La infraestructura es una escalera de cinco peldaños —hosting estático, serverless, PaaS (Platform as a Service, una plataforma que ejecuta tu app para que tú no tengas que gestionar el servidor), contenedores gestionados y VMs sin procesar— y deberías elegir el peldaño más simple que haga el trabajo.
- Prefiere los servicios **gestionados**: externaliza las operaciones (servidores, parches, escalado, respaldos) mientras conservas la comprensión.
- Deja que la IA escriba tu configuración de despliegue y tu workflow de CI (Continuous Integration, un robot automatizado que ejecuta tus verificaciones en cada push de código), pero léelo antes de confiar en él.
- “Gestionado” no es “mágico”: tus datos, tus secretos, tu factura y tu comprensión siguen siendo tu responsabilidad.
- Limita el gasto, confirma que HTTPS esté activo, y sabe cómo hacer rollback *antes* de lanzar.

Práctico

Toma un proyecto pequeño que ya tengas (incluso una sola página HTML) y desplégalo en un host estático. Luego pídele a la IA que genere un workflow de GitHub Actions que redespliegue en cada push, lee el YAML generado línea por línea, y confirma que puedes encontrar el botón de “volver al despliegue anterior” en el panel de control de tu host.

```
I have a static site in this repo. Walk me through deploying it to a free static host, then generate a GitHub Actions workflow that rebuilds and redeploys on every push to main. Explain each step of the YAML, and tell me where in the dashboard I roll back a bad deploy.
```

Infraestructura de bases de datos



Toda aplicación real termina necesitando recordar cosas: usuarios, pedidos, mensajes, configuraciones. Esa memoria vive en una base de datos. Para un principiante, “base de datos” puede sonar como una caja negra custodiada por expertos. No lo es. Este capítulo desmitifica la capa de datos para que puedas tomar buenas decisiones y dirigir a la IA para que haga el trabajo pesado de forma segura.

Lo que hay que interiorizar desde el principio: el código es reemplazable, los datos normalmente no lo son. Puedes reescribir una función con errores en una tarde, pero una columna que corrompiste el martes pasado —o una copia de seguridad que nunca hiciste— puede perderse para siempre. Esa asimetría es la razón por la que la capa de datos recompensa un poco de comprensión previa más que casi cualquier otra parte de tu stack.

Las tres formas de almacenar datos

La mayoría de las bases de datos caen en tres familias. No necesitas memorizar productos: necesitas reconocer qué *forma* encaja con tu problema.

Tipo	Ejemplos	Mejor para	Compensaciones
Relacional (SQL)	PostgreSQL, MySQL, SQLite	Datos estructurados con relaciones; cualquier cosa crítica en dinero o precisión	Debes diseñar un esquema de antemano; escalar escrituras muy grandes es más difícil
Documento / NoSQL	MongoDB, Firestore, DynamoDB	Datos flexibles o anidados, iteración rápida, formas de registro variadas	Fácil de crear datos inconsistentes; las relaciones y los informes se vuelven torpes
Clave-valor	Redis, Memcached, Cloudflare KV	Caché, sesiones, contadores, búsquedas rápidas por una clave conocida	Sin consultas por contenido; normalmente no es tu fuente de verdad

Las tres formas almacenan datos de maneras genuinamente distintas. Imagínalas lado a lado:

RELATIONAL (SQL)

rows + columns

id	email	name
1	a@...	Ana
2	b@...	Bob

tables reference each other

DOCUMENT (NoSQL)

self-contained blobs

```
{
  id, email,
  name,
  posts: [ ... ]
}
```

everything nested

KEY-VALUE

one key → one value

key	value
session	"x9f2"
count	"42"

no querying inside – look up by key only

Piensa en una base de datos relacional como un conjunto de hojas de cálculo que saben referenciarse entre sí. El lenguaje que usas para hablar con una de ellas es SQL (Structured Query Language, lenguaje de consulta estructurado). Una buena opción por defecto para el 90 % de los proyectos: **empieza con una base de datos relacional (Postgres)**. Es madura, predecible, impone estructura y maneja las relaciones que las aplicaciones reales inevitablemente desarrollan. Recurre a los almacenes de documentos cuando tus datos son genuinamente amorfos, y a los almacenes clave-valor como una capa de *apoyo* (caché), no como tu almacén principal.

La trampa que hay que evitar es elegir por intuición. “NoSQL (bases de datos que no usan tablas SQL: almacenan registros flexibles y de forma libre) escala mejor” es cierto solo para un conjunto reducido de problemas que la mayoría de las aplicaciones nunca enfrenta, e incluso entonces el precio es renunciar a las garantías de consistencia que hacen que los datos sean confiables. Elige la forma que encaje con tus *datos*, no la que suene más impresionante en un video de diseño de sistemas. Cuando tengas dudas, lo relacional es la apuesta segura: siempre puedes añadir una caché o un almacén de documentos después, pero deshacer un modelo de datos NoSQL enredado es doloroso.

Gestionado vs autoalojado

Puedes ejecutar una base de datos tú mismo en un servidor, o alquilar una **gestionada** (Supabase, Neon, PlanetScale, RDS, Cloudflare D1) donde el proveedor se encarga de las copias de seguridad, los parches y la disponibilidad.

Para la programación asistida por IA (vibe coding), **empieza siempre con una gestionada**. Autoalojar una base de datos significa que ahora eres responsable de las copias de seguridad, las actualizaciones de seguridad, el espacio en disco y la recuperación a las 3 de la madrugada. Eso es trabajo de operaciones real que distrae de construir. Los servicios gestionados cuestan un poco de dinero y ahorran una enorme cantidad de tiempo y riesgo. Pasa al autoalojamiento solo cuando tengas una razón concreta y las habilidades para respaldarla.

Cuando elijas un proveedor gestionado, vale la pena saber por qué estás pagando realmente, porque las páginas de marketing se mezclan entre sí. Algunas cosas genuinamente difieren:

- **Modelo de conexión.** Las plataformas sin servidor (Cloudflare Workers, funciones de Vercel) abren y cierran conexiones a la base de datos constantemente, lo que puede agotar un pool de conexiones clásico de Postgres. Proveedores como Neon y Supabase incluyen un *pooler de conexiones* (Pg-Bouncer o equivalente) exactamente para esto: usa la cadena de conexión con pooling en código sin servidor.
- **Ramificación y copias de seguridad.** Neon y PlanetScale te permiten “ramificar” la base de datos como en Git, de modo que puedes probar una migración en una copia. Confirma que la recuperación en un punto en el tiempo esté activada, y averigua hasta cuándo llega.
- **Dependencia del proveedor.** D1 y DynamoDB son excelentes pero te atan a una nube. Postgres plano (Supabase, Neon, RDS) es portable: tu esquema y tus consultas se mueven a cualquier host de Postgres. Para la mayoría de los principiantes, lo portable es la apuesta más segura a largo plazo.

Fundamentos del diseño de esquemas

Un *esquema* es el plano de tus datos: qué tablas existen, qué columnas tienen y cómo se relacionan. Un buen diseño de esquema es sobre todo sentido común más unos pocos hábitos.

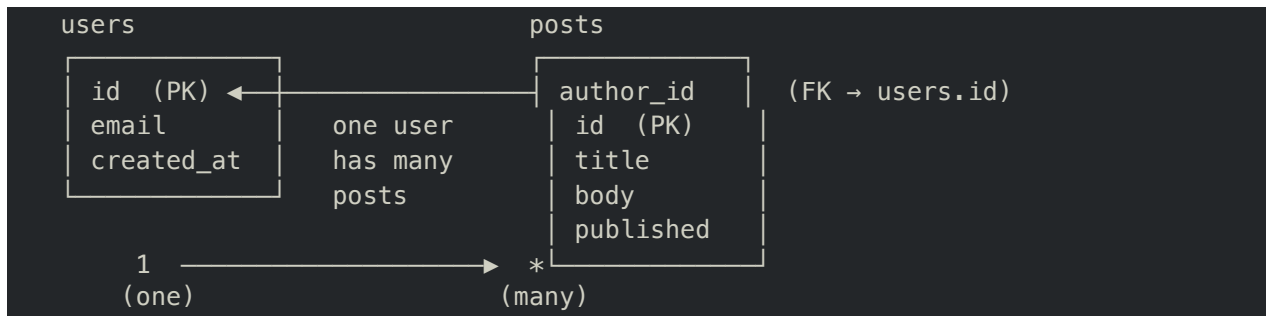
- **Una tabla por “cosa.”** Usuarios, publicaciones, pedidos: cada uno recibe su propia tabla.
- **Toda fila necesita un ID único** (una clave primaria).
- **Enlaza tablas con claves foráneas**, no copiando datos de un lado a otro.
- **Elige tipos honestos.** El dinero es un decimal, no un float. Las fechas son marcas de tiempo, no texto.
- **No dupliques.** Si el correo de un usuario vive en cinco tablas, tarde o temprano tendrás cinco correos distintos.

Aquí hay un esquema sencillo para un blog, del tipo que la IA generará por ti:

```
CREATE TABLE users (  
  id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),  
  email       TEXT NOT NULL UNIQUE,  
  created_at  TIMESTAMPTZ NOT NULL DEFAULT now()  
);  
  
CREATE TABLE posts (  
  id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),  
  author_id   UUID NOT NULL REFERENCES users(id),  
  title       TEXT NOT NULL,  
  body        TEXT NOT NULL,  
  published   BOOLEAN NOT NULL DEFAULT false,  
  created_at  TIMESTAMPTZ NOT NULL DEFAULT now()  
);  
  
CREATE INDEX idx_posts_author ON posts(author_id);
```

Esa línea `author_id ... REFERENCES users(id)` es la clave foránea. Conecta cada publicación con exactamente un usuario real, de modo que la propia base de datos se niega a almacenar una publicación

sin un autor válido:



Fíjate en REFERENCES (una clave foránea que ata una publicación a un usuario real) y en las restricciones NOT NULL y UNIQUE. Estas son barandillas que la base de datos impone por ti, de modo que los datos malos no puedan colarse aunque tu código tenga un error.

Una restricción que vale la pena añadir a propósito es qué sucede cuando el padre desaparece. Por defecto, eliminar un usuario cuyas publicaciones aún lo referencian fallará, que a menudo es lo que quieres. Pero puedes ser explícito: REFERENCES users(id) ON DELETE CASCADE elimina también las publicaciones, mientras que ON DELETE RESTRICT bloquea la eliminación. Elegir a propósito es mejor que descubrir el comportamiento por defecto de la manera difícil, en producción. El mismo instinto de “dejar que la base de datos te proteja” aplica a una restricción CHECK (CHECK (price >= 0)): rechaza datos sin sentido en el origen en vez de confiar en que cada ruta de código valide.

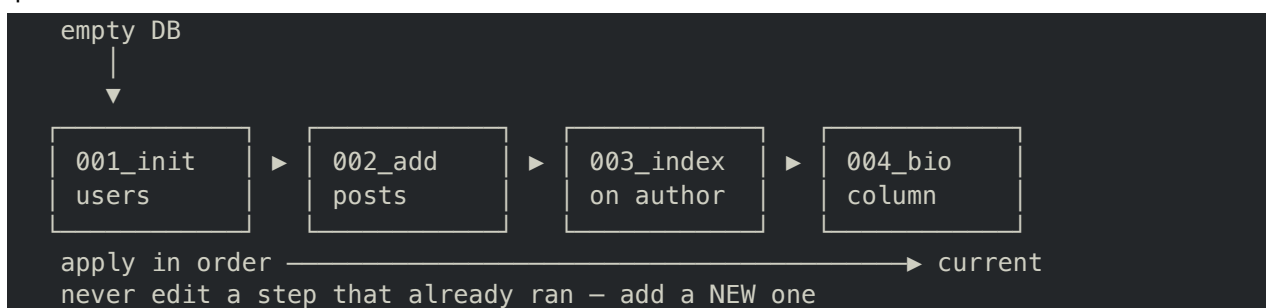
Migraciones: cambiar el esquema con seguridad

Tu esquema cambiará: añadirás una columna, renombrarás una tabla, agregarás un índice. Piénsalo como los cambios rastreados de un documento compartido: cada cambio queda registrado en orden, de modo que la copia de todos pueda actualizarse de la misma manera. Una **migración** es un script versionado y repetible que describe uno de esos cambios. Las migraciones importan porque el esquema en tu laptop, en la laptop de tu compañero de equipo y en producción debe permanecer idéntico.

Las reglas que mantienen las migraciones seguras:

- **Nunca edites la base de datos a mano en producción.** Escribe una migración.
- **Las migraciones avanzan hacia adelante.** Cada una es un archivo nuevo; no reescribes las antiguas que ya se ejecutaron.
- **Haz los cambios destructivos con cuidado.** Eliminar una columna o una tabla borra datos de forma permanente. Ten primero una copia de seguridad.
- **Los cambios aditivos son los más seguros.** Añadir una columna que admite nulos rara vez rompe algo; renombrar o eliminar una puede romper código en ejecución.

Las migraciones forman una cadena ordenada: cada una es un paso numerado que mueve cada copia de la base de datos de un estado conocido al siguiente, de modo que tu laptop, la de un compañero y producción terminan siendo idénticas:



Herramientas como Prisma, Drizzle o las migraciones de Rails generan y rastrean esto por ti. Déjalas hacerlo.

Una migración tiene dos mitades: el cambio a aplicar (up) y cómo deshacerlo (down). Una herramienta confiable genera ambas, de modo que un despliegue fallido pueda revertirse limpiamente:

```
-- up: add an optional bio, safe to apply live
ALTER TABLE users ADD COLUMN bio TEXT;

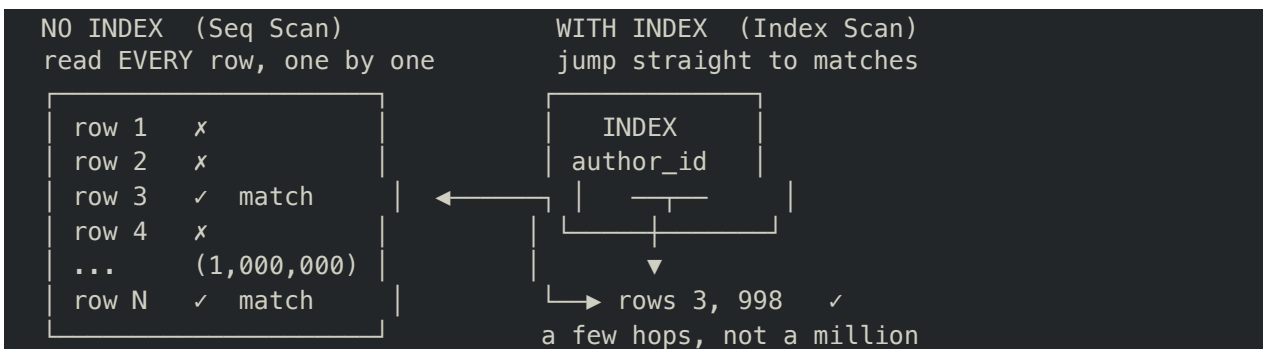
-- down: reverse it
ALTER TABLE users DROP COLUMN bio;
```

Las migraciones genuinamente peligrosas son los *renombrados* y los *cambios de tipo*, porque el código antiguo y el esquema nuevo discrepan brevemente. El patrón profesional es dividirlos en pasos aditivos: añade la columna nueva, rellénala con datos existentes (backfill), cambia el código para leerla y solo elimina la columna antigua en un despliegue *posterior*, una vez que nada la referencia. Se siente lento, pero es la diferencia entre un lanzamiento sin sobresaltos y un pico de errores de 500 páginas para cada usuario a mitad del despliegue.

Indexación

Un **índice** es como el índice al final de un libro: le permite a la base de datos saltar directamente a las filas coincidentes en lugar de escanear cada una. Sin índices, las consultas se vuelven más lentas a medida que los datos crecen: está bien con 100 filas, es doloroso con 1.000.000.

Sin un índice, la base de datos revisa cada fila una por una; con uno, salta directamente a la coincidencia, de la misma forma que el índice de un libro te envía a la página correcta en lugar de leerlo de principio a fin:



Las pautas prácticas:

- Indexa las columnas por las que **filtras o haces join** con frecuencia (p. ej. `author_id`, `email`).
- Las claves primarias se indexan automáticamente.
- **No indexes todo**: cada índice acelera las lecturas pero ralentiza las escrituras y ocupa espacio.
- Añade índices cuando veas una consulta lenta, no de forma preventiva para cada columna.

La forma honesta de averiguar si un índice ayuda es preguntarle a la base de datos, no adivinar. Todo motor SQL tiene EXPLAIN (en Postgres, EXPLAIN ANALYZE), que muestra el plan que va a usar:

```
EXPLAIN ANALYZE
SELECT * FROM posts WHERE author_id = '...';
```

Si la salida dice Seq Scan en una tabla grande, la base de datos está leyendo cada fila, una señal de que un índice ayudaría. Después de añadir uno, debería cambiar a un Index Scan. Esta es una tarea excelente para delegar a la IA: pega la consulta lenta y la salida de EXPLAIN, y pregunta qué índice añadir y por qué. Te mantienes en control comprendiendo la respuesta, no memorizando la teoría de índices.

Caché

Una **caché** almacena la respuesta a una pregunta costosa para que no tengas que volver a hacerla. Si tu página de inicio ejecuta la misma consulta pesada para cada visitante, guarda el resultado en caché en Redis durante 60 segundos y sívelo al instante.

El caché es una optimización de rendimiento, no una fuente de verdad. La trampa clásica de principiante es olvidar la **invalidación de caché**: la copia en caché se vuelve obsoleta y los usuarios ven datos antiguos. Regla general: cachea solo cuando hayas medido un cuello de botella real, y ten siempre una regla clara sobre cuándo expira o se refresca el valor en caché.

Hay dos estrategias limpias, y mezclarlas causa la mayoría de los errores de caché. **Expiración basada en tiempo (TTL — Time To Live, tiempo de vida, como una fecha de caducidad estampada en el valor cacheado)** dice “este valor es válido por 60 segundos, luego descártalo”: simple, y adecuado cuando unos datos ligeramente obsoletos son inofensivos (un contador de vistas, un feed de la página de inicio). **La invalidación basada en eventos** dice “elimina el valor cacheado en el momento en que cambian los datos subyacentes”: es correcta, pero debes recordar hacerlo en *cada* ruta de escritura, que es exactamente donde la gente falla. En caso de duda, prefiere un TTL corto: un valor que como máximo tiene un minuto de antigüedad es un error menor que un valor que está obsoleto para siempre porque olvidaste una invalidación.

Colas

Una **cola** te permite decir “haz esto después” en lugar de “haz esto ahora.” Cuando un usuario se registra, no quieres que espere mientras envías un correo de bienvenida y redimensionas su avatar. Empuja esos trabajos a una cola (RabbitMQ, BullMQ, Cloudflare Queues, SQS); un trabajador en segundo plano los recoge. El usuario obtiene una respuesta instantánea.

Usa una cola para cualquier tarea que sea lenta, que pueda fallar y reintentarse, o que no necesite terminar antes de responder al usuario.

El problema con el que tropiezan la mayoría de los principiantes es que un trabajo en cola puede ejecutarse *más de una vez*: un fallo de red hace que el trabajador reintente, y ahora el correo de bienvenida se envía dos veces. La solución es hacer que los trabajos sean **idempotentes**: diseñarlos de modo que ejecutar el mismo trabajo dos veces tenga el mismo efecto que ejecutarlo una vez (comprobar “¿ya envié esto?” antes de enviar). También decide de antemano qué pasa con un trabajo que sigue fallando; las colas maduras lo enrutan a una *cola de mensajes muertos* (dead-letter queue) para que un mensaje envenenado no bloquee todo lo que va detrás.

Errores comunes de principiante

- **Recurrir a NoSQL porque suena moderno.** La mayoría de las aplicaciones tienen relaciones; SQL las maneja mejor.
- **Almacenar dinero como punto flotante.** $0.1 + 0.2$ es famoso por no dar 0.3 . Usa tipos decimales.
- **Sin copias de seguridad.** Los servicios gestionados suelen respaldar automáticamente: confírmalo, no lo asumas.
- **Poner secretos o archivos enormes en la base de datos.** Los archivos pertenecen al almacenamiento de objetos (S3, R2); la base de datos almacena el *enlace*.
- **Dejar que la IA escriba consultas crudas con entradas del usuario pegadas directamente.** Esto causa inyección SQL. Usa consultas parametrizadas.
- **La consulta N+1.** Un bucle que ejecuta una consulta por elemento (obtener 100 publicaciones, luego 100 búsquedas separadas de autor) golpea a la base de datos. Obtén los datos relacionados en una sola consulta con un join, o agrupa las búsquedas.
- **Nunca hacer pruebas de carga.** Una consulta instantánea con las 50 filas de tu laptop puede arrastrarse con los 5 millones de producción. Genera una cantidad realista de datos ficticios antes de confiar en el rendimiento.

Dirigir a la IA para construir tu capa de datos

La IA es excelente para diseñar esquemas y escribir consultas... *si* le das las restricciones contra las que diseñar. No digas “haz una base de datos.” Describe el dominio, las relaciones y tus innegociables.

Un prompt sólido se ve así:

Design a PostgreSQL schema for a task-management app.

Entities and relationships:

- A user can own many projects.
- A project has many tasks.
- A task belongs to one project, has a status (todo / doing / done), and an optional due date.

Requirements:

- Use UUID primary keys and created_at timestamps.
- Enforce relationships with foreign keys.
- Add indexes for the columns we'll filter on.
- Give me the schema as SQL, plus a forward migration file.
- Explain any trade-offs you made.

Luego **revisa la salida** con la lista de comprobación de este capítulo: tipos honestos, claves foráneas y restricciones presentes, índices sensatos, sin datos duplicados. Cuando pidas consultas, exige explícitamente **consultas parametrizadas** para que la entrada del usuario nunca se concatene en el SQL. Y para cualquier migración que elimine o renombre algo, pídele a la IA que lo señale para que puedas hacer una copia de seguridad primero.

La base de datos es la parte de tu aplicación donde los errores son más difíciles de deshacer: los datos, una vez corrompidos o perdidos, no siempre vuelven. Precisamente por eso vale la pena conocer estos fundamentos: no para que escribas cada línea tú mismo, sino para que puedas dirigir a la IA con confianza y detectar los movimientos peligrosos antes de que se publiquen.

Recapitulación y práctica

Puntos clave

- La mayoría de las aplicaciones necesitan solo una de tres formas de almacenamiento —relacional, documento o clave-valor— y relacional es la opción segura por defecto.
- Empieza con una base de datos gestionada; tú conservas los datos y las decisiones de esquema, la plataforma mantiene los servidores en funcionamiento.
- Un buen diseño de esquema significa tipos honestos, claves foráneas y restricciones, índices sensatos, y sin datos duplicados.
- Trata cada migración como una puerta de un solo sentido: haz una copia de seguridad antes de cualquier cosa que elimine o renombre, y revisa de cerca las migraciones escritas por la IA.
- Exige siempre consultas parametrizadas para que la entrada del usuario nunca se concatene en el SQL.

Práctico

Elige una idea de aplicación pequeña y escribe primero sus entidades y relaciones en lenguaje sencillo. Luego entrégale esa descripción a la IA y pide un esquema más una migración hacia adelante, y revisa el resultado contra la lista de comprobación de este capítulo antes de ejecutar nada.

```
Design a relational schema for a personal bookmarks app.
```

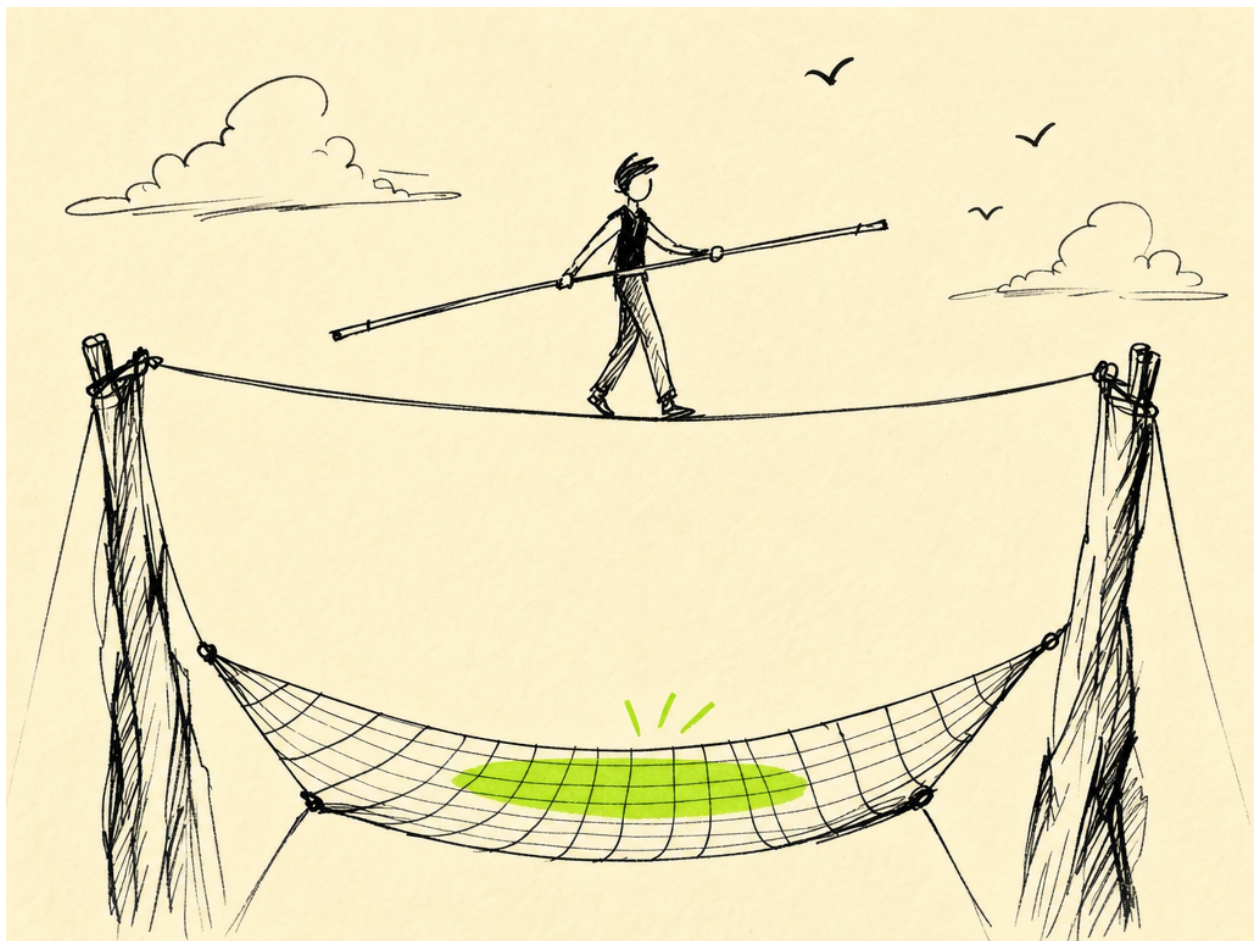
```
Entities and relationships:
```


- A user has many bookmarks.
- A bookmark belongs to one user, has a URL, a title, and many tags.
- A tag can apply to many bookmarks.

Requirements:

- Use UUID primary keys and created_at timestamps.
- Enforce relationships with foreign keys.
- Add indexes for the columns we'll filter on.
- Give me the schema as SQL plus a forward migration file.
- Flag any migration step that drops or renames so I can back up first.

Barreras de protección, pruebas y revisiones



La programación con IA (“vibe coding”) te permite avanzar rápido. El peligro es que también avanzas rápido en la dirección equivocada — y como la IA suena segura de sí misma, puede que no lo notes hasta que algo se rompa frente a un usuario real. Las barreras de protección son cómo mantienes la velocidad sin los accidentes.

Este capítulo no trata de convertir tu proyecto en una burocracia. Se trata de un puñado de comprobaciones baratas y automáticas que detectan los errores de la IA (y los tuyos) antes de que lleguen a producción. Configúralas una vez y funcionarán en segundo plano para siempre. La filosofía completa de este capítulo cabe en una línea: ***nunca publiques algo que no entiendes***. Las barreras de protección son lo que hace que “entenderlo” sea lo bastante barato como para que realmente lo hagas cada vez.

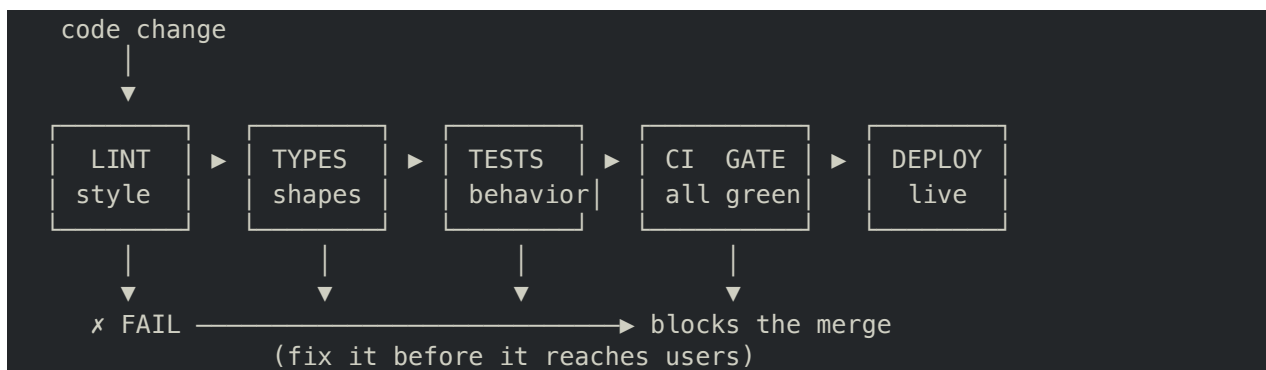
Por qué necesitas barreras de protección al dirigir a la IA

La IA no sabe lo que tu aplicación *debería* hacer. Detecta patrones y produce lo que parece plausible. Eso significa que hará alegremente cosas como:

- Eliminar una comprobación que no entiende “para arreglar el error”
- Refactorizar tres archivos cuando le pediste cambiar uno
- Introducir un error sutil que el camino feliz no revela
- Decirte con total confianza que ha terminado cuando nunca ejecutó el código

Las barreras de protección convierten el “confía en mí” en “demuéstralo.” El objetivo es un sistema donde sea *difícil* romper cosas por accidente, para que puedas seguir diciendo “solo pruébalo” sin contener la respiración.

Piensa en las barreras de protección como un pipeline por el que todo cambio debe pasar antes de llegar a los usuarios reales. Cada puerta es barata y automática; cualquiera de ellas puede detener en seco un cambio malo:

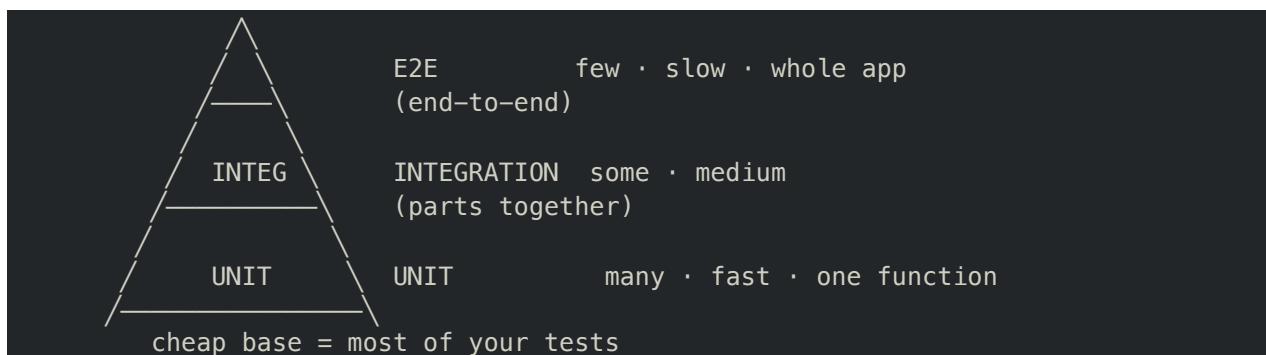


Hay una razón más profunda por la que esto importa más con IA que con un compañero humano. Un humano que elimina una comprobación de validación normalmente sabe que está asumiendo un riesgo, y siente el peso de esa decisión. La IA no siente nada. Eliminará una protección, reescribirá un archivo e informará del éxito con el mismo tono alegre, tanto si el cambio es brillante como catastrófico. No puedes leer su lenguaje corporal, porque no tiene ninguno. Las barreras de protección *son* su lenguaje corporal — son la única señal honesta que obtienes sobre si el trabajo realmente es sólido.

Pruebas: tu red de seguridad

Una prueba (test) es código que comprueba que otro código hace lo que esperas. Una vez escrita, se ejecuta en segundos y te avisa de inmediato si algo se ha roto.

Un conjunto de pruebas saludable tiene forma de pirámide: muchas pruebas unitarias pequeñas y rápidas en la base, menos pruebas de integración en el medio, y solo un puñado de pruebas de extremo a extremo lentas en la cima. La mayor parte de tu seguridad viene barata desde abajo:



No necesitas una cobertura del 100 %. Empieza probando las cosas que más dolerían si se rompieran: inicio de sesión, pagos, datos que no se pueden regenerar, lógica de negocio central. La IA es excelente escribiendo pruebas — a menudo mejor que escribiendo funcionalidades, porque el comportamiento esperado es concreto.

Aquí tienes un prompt que funciona bien:

```
Write tests for the calculateDiscount function in src/pricing.js.  
Cover: a normal case, the zero/empty case, the maximum-discount  
boundary, and one invalid input. Use the existing test setup in  
this repo (check how other *.test.js files are structured).  
Run the tests and show me they pass before you finish.
```

Esa última línea importa. Pide siempre a la IA que *ejecute* las pruebas, no solo que las escriba. Una prueba que nunca se ejecutó es una suposición. También vale la pena ser específico sobre *qué* casos cubrir, porque dejada a su aire la IA tiende a escribir tres variaciones del camino feliz y llamarlo exhaustivo. Los casos que realmente detectan errores son los aburridos: entrada vacía, el valor límite, lo que debería lanzar una excepción.

Un ejemplo sencillo de lo que debería producir:

```
import { calculateDiscount } from "../pricing.js";  
import { describe, it, expect } from "vitest";  
  
describe("calculateDiscount", () => {  
  it("applies a normal percentage discount", () => {  
    expect(calculateDiscount(100, 0.2)).toBe(80);  
  });  
  
  it("returns the full price when discount is zero", () => {  
    expect(calculateDiscount(100, 0)).toBe(100);  
  });  
  
  it("never discounts below zero", () => {  
    expect(calculateDiscount(100, 1.5)).toBe(0);  
  });  
  
  it("rejects a negative price", () => {  
    expect(() => calculateDiscount(-10, 0.2)).toThrow();  
  });  
});
```

Cuando más adelante cambies una funcionalidad, ejecuta primero las pruebas. Si se ponen en rojo, en-contraste la rotura antes que tus usuarios. Cuando la IA arregle un error, pídele que añada una prueba que falle con el comportamiento antiguo — así el error no puede volver silenciosamente. Este hábito tiene un nombre: la prueba de regresión. Cada error que encuentras es una lección que el código ya pagó; una prueba de regresión es cómo conservas la lección.

Una advertencia: una prueba también puede estar equivocada. Si la IA escribe una prueba que afirma que el comportamiento *defectuoso* es correcto, pasará para siempre mientras tu aplicación sigue rota. Así que lee las aserciones, no solo las marcas verdes. Una prueba que pasa solo significa “el código hace lo que dice la prueba” — todavía tienes que confirmar que la prueba dice lo correcto.

Comprobación de tipos y linting: errores gratis detectados

Dos herramientas detectan categorías enteras de errores con casi ningún esfuerzo por tu parte:

- La **comprobación de tipos** (TypeScript, mypy, etc.) detecta “pasaste una cadena de texto donde se esperaba un número” y “este valor podría ser undefined.” A la IA le encanta olvidar casos límite que los tipos hacen imposibles.
- El **linting** (ESLint, Ruff, etc.) detecta variables sin usar, código inalcanzable y desviaciones de estilo. Mantiene un código base tocado por IA sin que poco a poco se convierta en un desastre.

Configura esto pronto y dile a la IA que lo obedezca: *“Ejecuta el comprobador de tipos y el linter, y arregla todo lo que reporten antes de dar esto por terminado.”* Trata una comprobación de tipos limpia como parte de la definición de “terminado.”

Sin embargo, vigila las salidas fáciles y perezosas. Cuando el comprobador de tipos se queja, la solución genuinamente correcta es manejar el caso que encontró — el valor undefined, el tipo incorrecto. La solución *rápida* es silenciar la queja con `// @ts-ignore`, `any`, o un comentario `# type: ignore`, y la IA recurre a esto constantemente porque hace desaparecer el texto rojo. También hace que la comprobación sea inútil justo donde intentaba ayudar. Si ves aparecer un comentario de supresión en un diff, trátalo como una pregunta, no como una solución: *“¿Por qué es necesario esto? Arregla el tipo subyacente en lugar de ignorarlo.”*

Delimitar el alcance de los cambios de la IA

La mayor causa de daño inducido por IA es dejar que cambie demasiado a la vez. Un diff de 400 líneas repartido en nueve archivos es imposible de revisar y fácil para esconder un error.

Mantén los cambios pequeños y contenidos:

- Pide **un cambio lógico a la vez**, no “ya que estás, también...”
- Dile explícitamente: *“Solo toca `src/auth.js`. No refactorices nada más.”*
- Confirma (commit) estados que funcionan a menudo, para que siempre tengas un punto limpio al que volver.
- Cuando proponga una gran reescritura, pídele que divida el trabajo en pasos que puedas revisar uno por uno.

Un alcance pequeño hace posible el siguiente paso — leer el diff — de verdad. Aquí hay un efecto compuesto: los commits pequeños también tienen un radio de explosión pequeño. Si un cambio de alcance ajustado resulta estar mal, revocas un commit y pierdes diez minutos. Si un cambio extenso está mal, o bien te pasas una hora desenredando qué parte rompió las cosas, o tiras a la basura una tarde entera de trabajo. La disciplina de alcance es un seguro barato que compras antes de saber que lo necesitarás.

Leer diffs con espíritu crítico

Un diff es la lista exacta de lo que cambió: líneas rojas eliminadas, líneas verdes añadidas. Leer diffs es el hábito de mayor impacto en la programación con IA, porque es donde detectas a la IA haciendo algo que no pediste.

No tienes que entender cada línea. Busca estas señales de alarma:

- **Archivos que no esperabas que cambiaran.** ¿Por qué se tocó un archivo de configuración?
- **Código eliminado.** ¿Eliminó una comprobación de validación, un manejador de errores o una prueba “para que pasara”?
- **Secretos o valores codificados directamente.** Claves de API (Application Programming Interface — interfaz de programación de aplicaciones), contraseñas, o URLs (Uniform Resource Locators — direcciones web) que no deberían estar en el código.
- **Comprobaciones desactivadas.** Una prueba omitida, una protección comentada, un `// @ts-ignore`, o un permiso relajado.
- **Alcance descontrolado (scope creep).** Cambios muy alejados de lo que pediste.

Si algo parece raro, pregunta: “*Explica por qué cambiaste X — no pedí eso.*” Haz que la IA justifique especialmente las eliminaciones. “Estaba causando un error” es una razón para entender el error, no para eliminar la comprobación.

Una forma práctica de construir el hábito: lee el diff *antes* de aceptar el cambio, nunca después. Una vez que el código está fusionado y la aplicación parece funcionar, nunca volverás atrás para leerlo — no hay fricción que te obligue. El momento de la revisión tiene que ser el momento de la decisión. Las herramientas ayudan aquí: `git diff --stat` te da un resumen de una línea por archivo para que un archivo inesperado salte a la vista de inmediato, y revisar en la página de un pull request muestra todo el cambio en una vista desplazable en lugar de disperso por tu editor.

Puertas de revisión: una lista de comprobación antes de fusionar

Una puerta de revisión (review gate) es una lista corta de cosas que deben ser ciertas antes de que un cambio se publique. Repásala tú mismo, o haz que la IA confirme cada elemento:

- ☐ Leí el diff y entiendo qué cambió y por qué
- ☐ El cambio está delimitado a lo que realmente pedí
- ☐ Las pruebas pasan (y el comportamiento nuevo tiene una prueba)
- ☐ La comprobación de tipos y el linter están limpios
- ☐ No quedaron secretos, registros de depuración ni código comentado
- ☐ No se eliminó silenciosamente ninguna comprobación de seguridad
- ☐ Probé el cambio yo mismo, al menos el camino feliz

Si no puedes marcar una casilla, no has terminado.

La lista de comprobación es deliberadamente corta porque una puerta que te saltas no es ninguna puerta. La tentación, especialmente cuando avanzas rápido y el cambio “se ve bien,” es dejarlo pasar sin más. Resiste eso en cualquier cosa de cara al usuario. Un truco útil es hacer que la IA la rellene por ti — pega la lista y pregunta: “*Repasa esta puerta de revisión para el cambio que acabas de hacer. Para cada elemento, di PASS (aprobado) o FAIL (fallido) con una línea de evidencia.*” No puede escribir honestamente “las pruebas pasan” sin haberlas ejecutado, y obligarla a citar evidencia por línea convierte un vago “se ve bien” en algo que puedes auditar.

Fundamentos de CI: haz que las comprobaciones sean automáticas

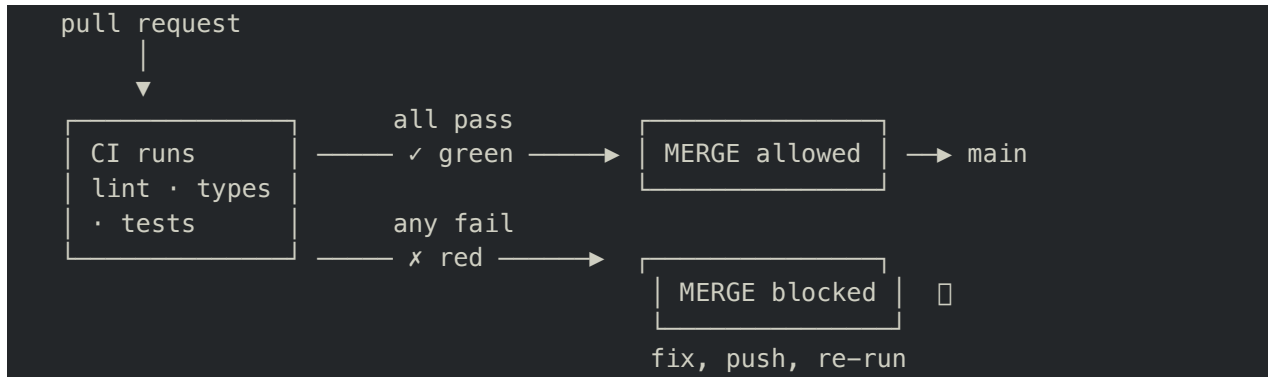
CI (Continuous Integration — integración continua) es un robot que ejecuta tus comprobaciones cada vez que subes código, para que no puedas *olvidarte* de ejecutarlas. La mayoría de las plataformas lo reducen a un único archivo. Aquí tienes un ejemplo mínimo de GitHub Actions:

```
# .github/workflows/ci.yml
name: CI
on: [push, pull_request]

jobs:
  check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 20
      - run: npm ci
      - run: npm run lint
      - run: npm run typecheck
      - run: npm test
```

Ahora cada push se somete a linting, comprobación de tipos y pruebas automáticamente. Si algo falla, ves una marca roja antes de que el código se pueda fusionar. Incluso puedes pedirle a la IA que escriba este archivo para tu stack — “Añade un flujo de trabajo de GitHub Actions que ejecute nuestro lint, comprobación de tipos y pruebas en cada pull request.”

Con la protección de ramas activada, una comprobación en rojo no solo te avisa — bloquea físicamente el botón de fusión hasta que la compilación vuelva a estar en verde:



El objetivo de la CI no es la automatización por sí misma — es que saca las comprobaciones de tu memoria y las mete en el pipeline, donde no se pueden saltar en una noche cansada. Vale la pena hacer dos mejoras una vez que lo básico funcione en verde. Primero, convierte las comprobaciones en un estado *obligatorio* en tu rama principal (una “regla de protección de rama” en GitHub) para que una compilación en rojo bloquee literalmente el botón de fusión en lugar de mostrar solo una advertencia que puedes ignorar. Segundo, cuando la CI falle, pega el registro (log) del fallo directamente a la IA: “La CI falló en esta ejecución, aquí está la salida — diagnostica y arréglalo.” El registro es un informe de error preciso, generado por máquina, que es exactamente el tipo de entrada con la que la IA actúa mejor.

Hábitos seguros por defecto

Reúnelo todo en reflejos que te permitan avanzar rápido sin miedo:

- **Trabaja en una rama, no directamente en producción.** Las ramas son gratis; la producción rota no lo es.
- **Confirma (commit) estados pequeños y funcionales a menudo.** Los commits frecuentes hacen que revertir sea trivial.
- **Pide siempre a la IA que ejecute lo que escribió** — las pruebas, la aplicación, el comprobador de tipos.
- **Mantén una terminal para la IA y otra para ti**, para poder verificar de forma independiente.
- **Nunca dejes que se publique un cambio que no leíste.** La velocidad viene de buenas barreras de protección, no de saltarse la revisión.

Las barreras de protección no te frenan — son lo que te permite decir “sí, publícalo” sin un nudo en el estómago. Configúralas una vez, y la IA puede correr a toda velocidad mientras la red de seguridad amortigua las caídas.

Cuando una barrera de protección grita: haz triaje antes de arreglar

Las barreras de protección solo sirven si respondes bien a ellas, y el patrón de fallo es predecible: la compilación se pone en rojo, entras en un poco de pánico, y le pegas el error a la IA con un “arréglalo.” A veces funciona. A menudo produce un arreglo que hace desaparecer el texto rojo sin abordar lo que realmente estaba mal — la IA debilita la prueba, amplía un tipo, o envuelve la llamada que falla en un try/catch que traga el problema.

Antes de dejar que “arregle” algo, dedica diez segundos al triaje. En realidad solo hay tres tipos de fallo de barrera de protección:

- **Un error real que acabas de introducir.** La comprobación está haciendo su trabajo. Arregla el código, no la comprobación.
- **Una prueba que ahora está desactualizada.** Cambiaste el comportamiento a propósito, y la aserción antigua ya no coincide. Actualiza la prueba para afirmar el *nuevo* comportamiento correcto — de forma deliberada, no reflexiva.
- **Una comprobación inestable (flaky) o incorrecta.** Genuinamente raro. Sé desconfiado contigo mismo aquí; este es el cajón donde la gente tira errores reales porque es el que les permite seguir adelante.

La instrucción que mantiene a la IA honesta es hacer que diagnostique antes de cambiar nada:

```
The test "never discounts below zero" is failing.  
First, tell me WHY it fails – what does the code do now vs. what  
the test expects. Then tell me which is correct: the code or the  
test. Do NOT change anything until I confirm.
```

Esa única pausa es la diferencia entre una barrera de protección que te protege y una barrera de protección que has entrenado silenciosamente a la IA para derrotar. Una comprobación en rojo es información; lo peor que puedes hacer es pagar por ponerla en verde sin leer lo que te estaba diciendo.

Repaso y práctica

Conclusiones clave

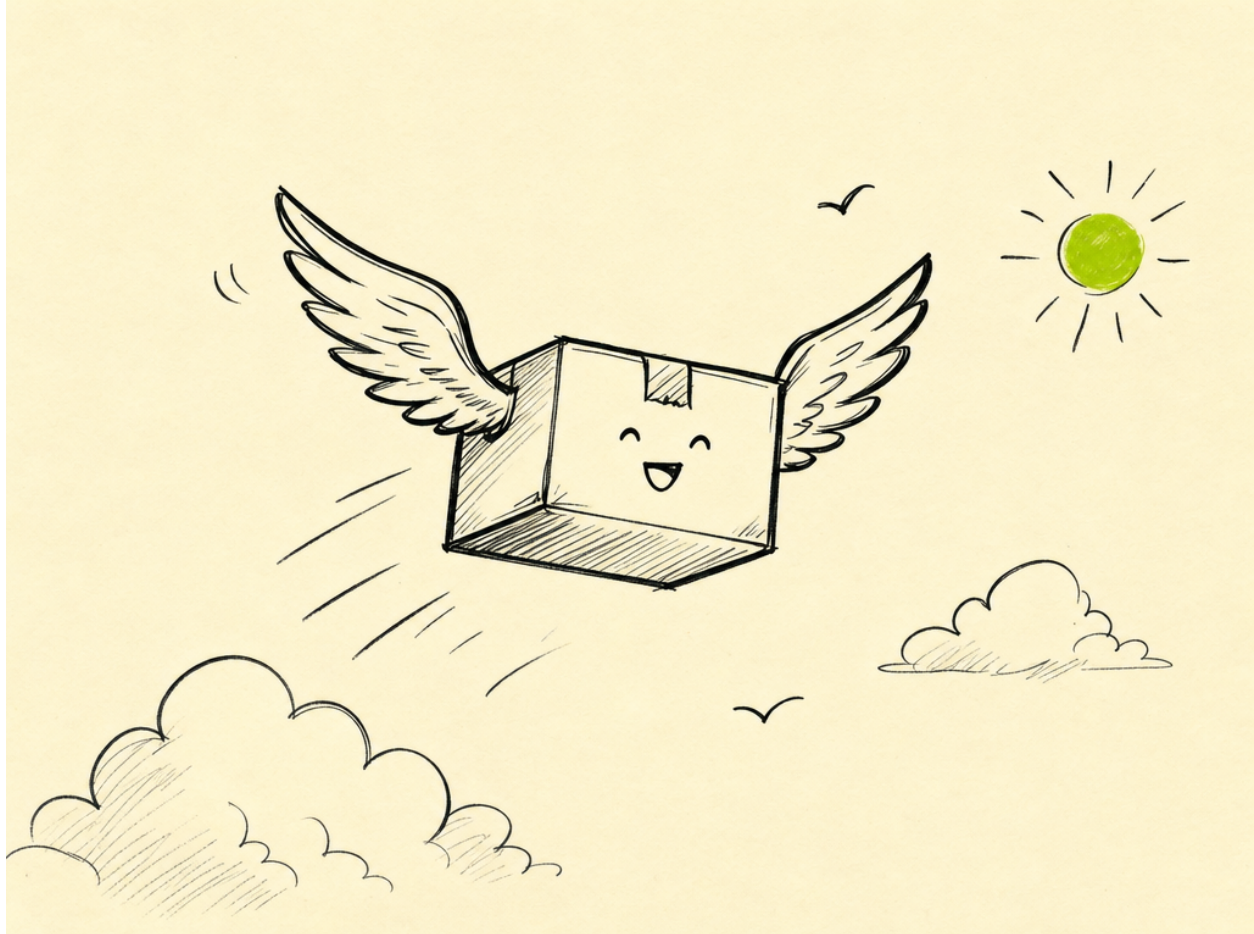
- Las barreras de protección son lo que permite a la IA correr a toda velocidad mientras una red de seguridad amortigua las caídas — te aceleran, no te frenan.
- Las pruebas, la comprobación de tipos y el linting detectan errores gratis; conéctalos a la CI para que las comprobaciones se ejecuten automáticamente.
- Delimita estrechamente los cambios de la IA y lee con espíritu crítico cada diff antes de fusionar.
- Una comprobación en rojo es información — haz triaje (error real / prueba desactualizada / comprobación inestable) antes de dejar que la IA lo “arregle.”
- Nunca publiques un cambio que no leíste.

Prácticalo

En uno de tus proyectos, configura una comprobación automatizada que aún no tengas — una sola prueba, un linter, o un paso de comprobación de tipos — y añádela a un flujo de trabajo de CI para que se ejecute en cada push. Luego rompe deliberadamente algo pequeño y confirma que la comprobación se pone en rojo antes de arreglarlo.

```
Add a CI workflow to this repo that runs the test suite, the linter,  
and the type checker on every push and pull request. Generate the  
config, explain each step, and tell me how to read the output when a  
check fails so I can tell a real bug from an out-of-date test.
```

Publicar en producción



Has estado construyendo. La app funciona en tu máquina, la demo dejó boquiabierto a un amigo, y las ganas de sacarla al mundo son reales. Este es el momento en que muchos creadores en solitario se atascan — no porque el código esté roto, sino porque “salir a producción” se siente como una disciplina distinta con sus propias artes oscuras: DNS (Domain Name System, el sistema de nombres de dominio — la agenda de direcciones de internet que convierte un nombre como `tudominio.com` en el número donde vive un servidor), servidores, secretos, webhooks de pago. Buenas noticias: este es exactamente el tipo de trabajo mecánico y bien documentado en el que la IA es mejor. Tu trabajo es dirigir. El suyo es escribir los comandos.

Pero “dirigir” es la palabra que sostiene todo. El tema de todo este libro es *entender lo que envías*. No necesitas memorizar los flags, pero sí necesitas saber para qué sirve cada paso — qué te protege un webhook, por qué un secreto nunca debe llegar a tu repositorio, qué está enjaulando un límite de gasto. Cuando entiendes la forma de la cosa, puedes notar cuándo la respuesta segura de la IA está silenciosamente equivocada, y puedes recuperarte cuando el primer despliegue explota. Los creadores que se se queman en el lanzamiento son casi siempre los que pegaron comandos que no podían leer.

Este capítulo recorre toda la última milla, desde una app que funciona en local hasta una URL (Uniform Resource Locator, la dirección web que la gente escribe en un navegador) real por la que la gente puede pagarte.

Elige un host y despliega

Para la mayoría de las apps hechas con vibe coding, no necesitas un servidor que tengas que cuidar tú mismo. Las plataformas modernas toman tu código y te dan una URL en vivo en minutos. Las opciones comunes:

- **Sitios estáticos y frontends:** Cloudflare Pages, Vercel, Netlify, GitHub Pages.
- **Apps full-stack y APIs (Application Programming Interfaces, las puertas que un programa usa para hablar con otro):** Cloudflare Workers, Vercel, Render, Fly.io, Railway.
- **Backends más pesados:** una plataforma de contenedores o un pequeño VPS (Virtual Private Server, tu propia porción alquilada de un ordenador siempre encendido en un centro de datos).

Lo correcto es contarle a la IA qué construiste y dejar que ella recomiende. Prueba: *“Esto es una app de Next.js con una base de datos Postgres. Quiero el host más barato y confiable con un plan gratuito generoso. Dame los pasos de despliegue y un archivo de configuración.”* Sé específico sobre las cosas que realmente condicionan la elección: tu framework, si tienes base de datos, si necesitas trabajos en segundo plano o subida de archivos, y tu presupuesto. Una página estática de marketing y una app full-stack con una cola de tareas tienen hogares muy distintos.

Una vez elegido, desplegar suele ser un par de comandos:

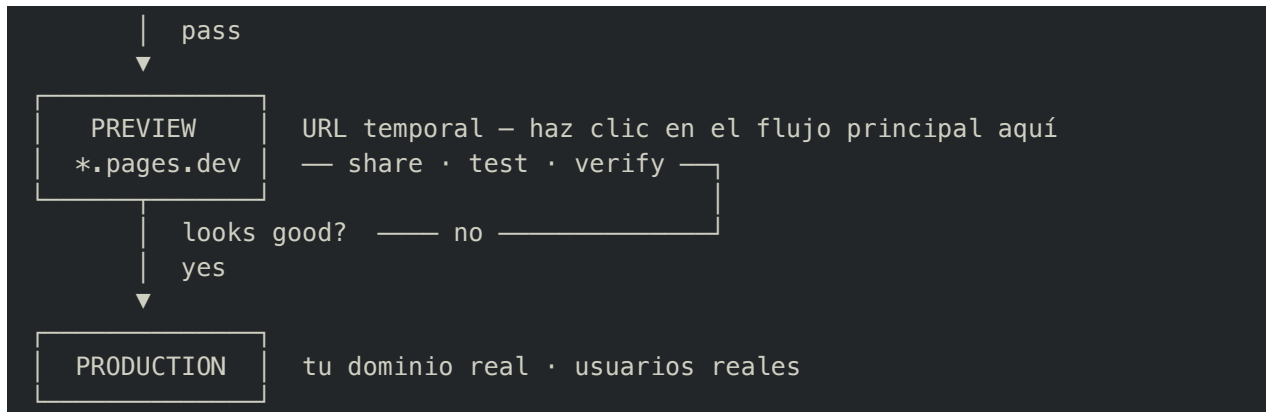
```
# Ejemplo: Cloudflare Workers vía Wrangler
npm install -g wrangler
wrangler login
wrangler deploy

# Ejemplo: Vercel
npm install -g vercel
vercel --prod
```

Deja que la IA genere la configuración de la plataforma (`wrangler.toml`, `vercel.json`, etc.) y que explique cualquier error que arroje el primer despliegue. Los primeros despliegues casi siempre arrojan algo — un comando de build que falta, un directorio de salida equivocado, un desajuste de versión de Node, una variable de entorno (un ajuste con nombre que se le entrega a tu app desde fuera del código, como una llave bajo el felpudo que la app sabe que debe buscar) que el build necesita pero no encuentra. Nada de esto significa que hiciste algo estúpido; significa que hay que decirle a la plataforma una cosa más. Pega el error tal cual y pide la solución, y pregunta *por qué* pasó para que el próximo te sorprenda menos. Dos o tres rondas y ya estás en vivo en una URL temporal. Abre esa URL y recorre el flujo principal antes de celebrar — “el despliegue tuvo éxito” y “la app funciona” no son la misma afirmación.

Cada despliegue recorre el mismo pipeline. El paso de previsualización es tu red de seguridad: le da a cada cambio su propia URL para hacer clic antes de que pueda tocar a usuarios reales.





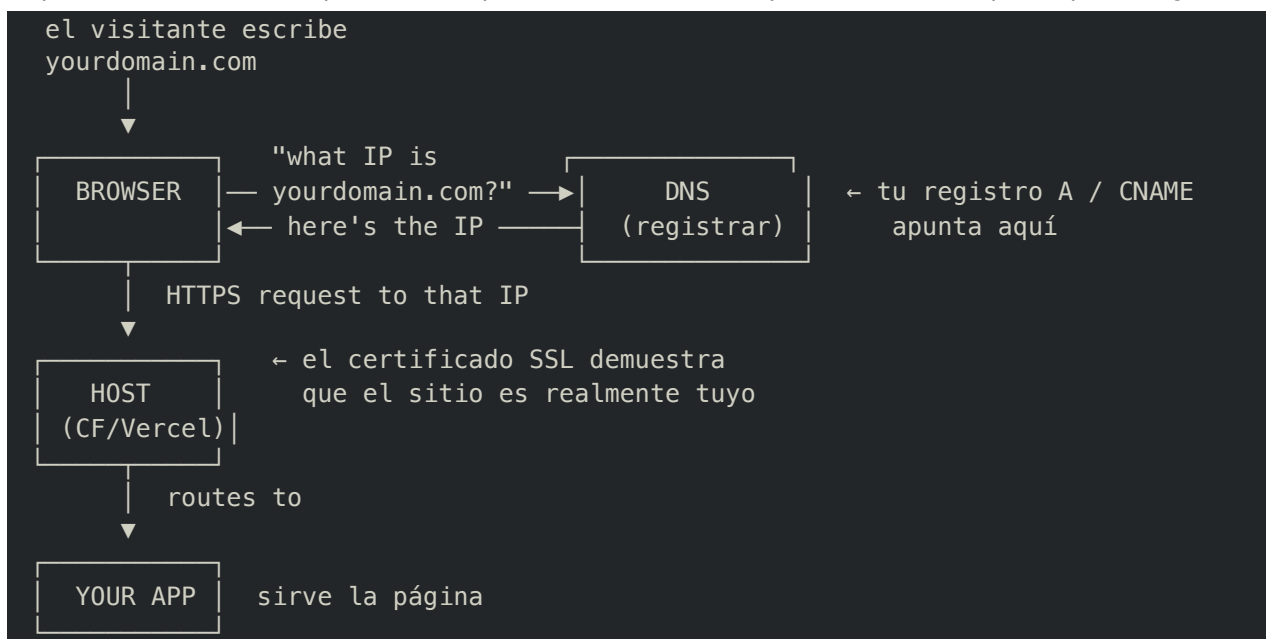
Dominios personalizados y DNS

Una URL `*.workers.dev` o `*.vercel.app` está bien para probar, pero quieres tu propio nombre. El flujo:

1. Compra el dominio (Namecheap, Cloudflare Registrar, Porkbun — elige uno).
2. Apunta su DNS hacia tu host. Esto suele significar agregar un registro CNAME o A, o cambiar por completo los nameservers del dominio.
3. Agrega el dominio en el panel de tu host y deja que emita un certificado SSL (la tarjeta de identidad digital que demuestra que el sitio es realmente tuyo y activa el candado y HTTPS — la versión segura y cifrada del protocolo de transferencia de la web) automáticamente.

Un modelo mental rápido para que los registros dejen de sentirse como magia: un **registro A** apunta un nombre directamente a una dirección IP; un **CNAME** apunta un nombre hacia *otro nombre* (útil cuando la IP del host puede cambiar bajo tus pies); y los **nameservers** entregan el dominio completo a un proveedor para que gestione cada registro por ti. La mayoría de los hosts modernos prefieren un CNAME o un cambio completo de nameservers — quieren gestionar el certificado SSL y el enrutamiento en el borde ellos mismos.

Aquí está toda la cadena que recorre la petición de un visitante, y dónde vive cada pieza que configuraste:



El DNS es la parte más confusa para principiantes, sobre todo por la jerga y los retrasos de propagación.

Deja que la IA lo traduzca. Pregunta: “*Mi dominio está registrado en Namecheap y estoy alojando en Cloudflare Pages. Guíame exactamente qué registros agregar, con qué valores, y cómo verificar que funcionó.*” Luego verifica el resultado tú mismo con una comprobación rápida en lugar de solo refrescar el navegador:

```
# Consulta los registros que el mundo realmente ve
dig yourdomain.com
nslookup yourdomain.com

# Confirma que el certificado está activo y no ha expirado
curl -I https://yourdomain.com
```

Los cambios de DNS pueden tardar desde minutos hasta un día en propagarse, y la caché de tu propio navegador es la peor mentirosa de todas — te mostrará con gusto una respuesta obsoleta mucho después de que el resto de internet se haya actualizado. Si no está activo de inmediato, eso es normal: revisa desde tu teléfono con datos móviles, o usa un verificador de “propagación DNS” en línea, antes de asumir que algo está roto.

Variables de entorno y secretos

Tu código necesita claves de API, URLs de base de datos y tokens. Estos **nunca** deben subirse a tu repositorio. Una vez que un secreto está en el historial de git, es efectivamente público para siempre — borrar la línea en un commit posterior *no* lo elimina del historial, y hay bots que rastrean repositorios públicos nuevos en busca de claves filtradas en cuestión de minutos. Dos reglas te llevan lejos:

- Guarda los secretos en un archivo `.env` local, y asegúrate de que `.env` esté en tu `.gitignore`.
- Configura los mismos valores como variables de entorno en el panel de tu host (o vía CLI — la Command-Line Interface, la ventana de comandos de texto donde escribes en lugar de hacer clic) para producción.

```
# .env (solo local — nunca subir esto)
DATABASE_URL=postgres://localhost:5432/app
STRIPE_SECRET_KEY=sk_test_xxx
SESSION_SECRET=change-me

# Configura los equivalentes de producción en el host
wrangler secret put STRIPE_SECRET_KEY
vercel env add STRIPE_SECRET_KEY production
```

Mantén un `.env.example` versionado que liste los *nombres* (no los valores) de cada variable, para que tú en el futuro y la IA sepan qué espera la app. Fíjate en el prefijo `sk_test_` de arriba — mantén tus claves de prueba y en vivo claramente diferenciadas, y nunca dejes que una clave en vivo se quede en un `.env` local que podrías pegar en un chat. Si alguna vez pegas un secreto en un chat o lo subes por accidente, trátalo como comprometido y róvalo de inmediato: genera una clave nueva en el panel del proveedor, actualízala en todos lados, y revoca la anterior. Rotar te cuesta cinco minutos; una URL de base de datos filtrada puede costarte todo lo que hay en la base de datos.

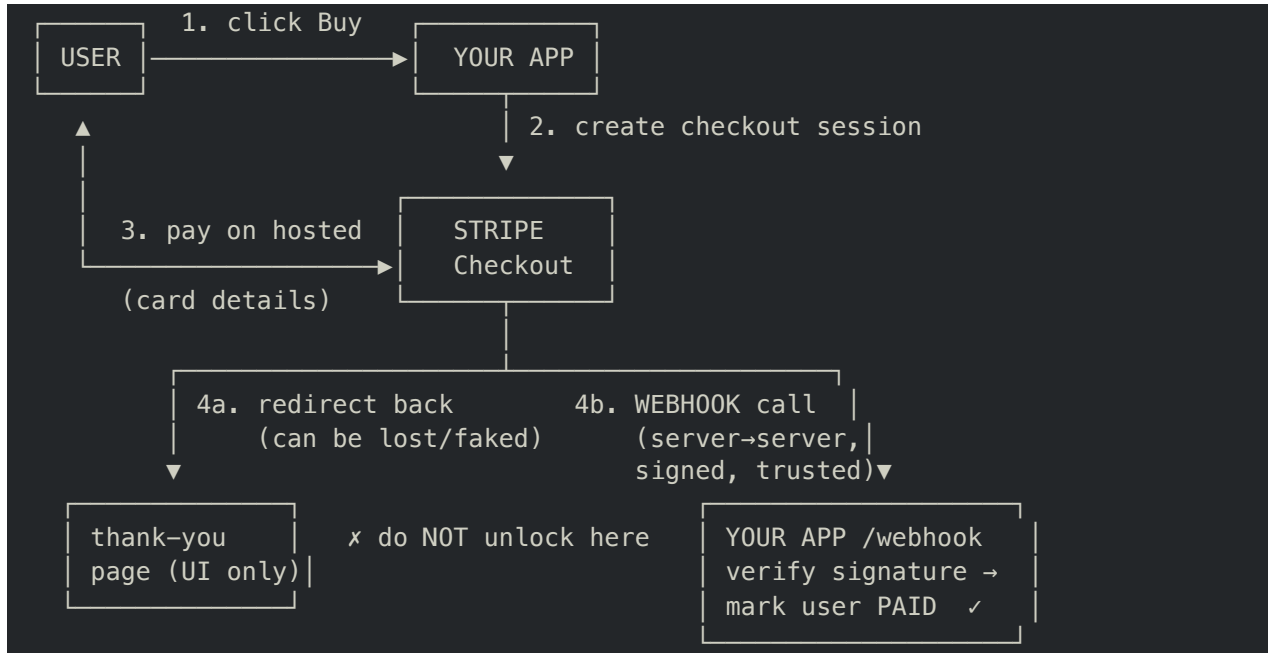
Conectando los pagos

Si quieres cobrar, el camino estándar para un creador en solitario es Stripe (o Lemon Squeezy / Paddle si quieres que ellos gestionen el impuesto sobre las ventas por ti — como merchant of record, se quedan con una comisión mayor pero hacen desaparecer el dolor de cabeza fiscal). La mecánica es repetitiva y está bien documentada, lo que la hace ideal para delegar. Las piezas que necesitas:

- Un flujo de **checkout** — Stripe Checkout es el más rápido; te entrega una página de pago alojada por ellos para que nunca toques datos de tarjeta en bruto.

- Piensa en un webhook como la empresa de pagos llamando a tu app en el instante en que se completa una venta, en lugar de que tu app tenga que estar preguntando “¿ya se completó?”. Un **webhook** es exactamente eso: un endpoint (una URL en tu servidor) que el proveedor de pagos llama para que tu app sepa cuándo un pago realmente tuvo éxito. Esta es la parte que los principiantes se saltan y luego se preguntan por qué los pedidos nunca se completan. La redirección de vuelta a tu sitio puede cerrarse, bloquearse o falsificarse; el webhook es la fuente de verdad servidor a servidor. Nunca confíes solo en la redirección; confía en el webhook.
- Una forma de **almacenar los derechos** — marcar al usuario como pagado en tu base de datos, para que un refresco de página o un dispositivo nuevo sigan viendo lo que compró.

La idea clave es *cuál* flecha confías. La redirección del usuario de vuelta a tu sitio puede perderse o falsificarse; el webhook es la línea servidor a servidor que realmente desbloquea el producto:



Pídele a la IA que estructure las tres piezas y que explique la verificación de la firma del webhook, ya que ese es el único paso sensible en cuanto a seguridad — sin ella, cualquiera que encuentre la URL de tu webhook podría falsificar un evento de “pago exitoso” y desbloquear tu producto gratis. Prueba todo en el modo de prueba de Stripe con sus números de tarjeta de prueba (4242 4242 4242 4242 es el famoso) antes de pasar a claves en vivo, y usa la CLI de Stripe para reenviar eventos de webhook a tu máquina local y así poder verlos dispararse. No te saltes el ensayo general en modo de prueba. El primer dinero real que pase por tu app debería ser una transacción que ya has visto funcionar una docena de veces.

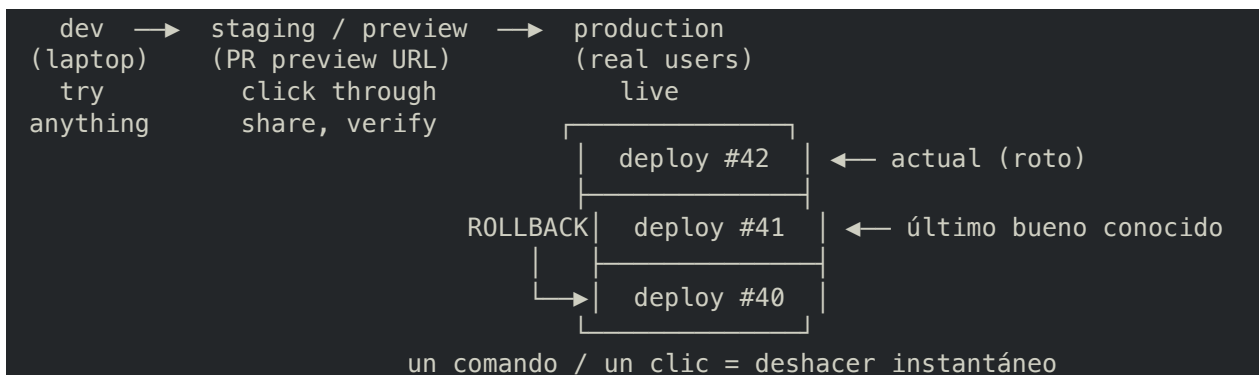
Staging y reversiones

Dos hábitos separan un lanzamiento que se recupera con elegancia de uno que se convierte en pánico a medianoche: probar los cambios en algún lugar seguro antes de que lleguen a usuarios reales, y poder deshacer un mal despliegue al instante.

Un **entorno de staging** es simplemente una segunda copia de tu app — el mismo código, una URL separada, una base de datos separada (o desechable) — donde pruebas cosas antes de producción. La mayoría de los hosts hacen esto casi gratis: cada pull request obtiene su propia **URL de previsualización** automáticamente, así puedes recorrer un cambio y compartirlo con alguien antes de fusionarlo. Úsalo especialmente para cualquier cosa que no puedas revertir fácilmente, como una migración de base de datos.

Imagina los tres entornos como una escalera que un cambio sube, con la reversión como la cuerda que

hace caer producción directamente al último peldaño bueno:



Un **rollback** (reversión) es tu botón de deshacer para toda la app. Como estas plataformas conservan cada despliegue anterior, volver a la última versión buena suele ser un comando o un clic en el panel:

```
# Cloudflare Pages: redesplegar un build conocido como bueno
wrangler pages deployment list
wrangler pages deployment rollback <DEPLOYMENT_ID>

# Vercel: promover un despliegue anterior de vuelta a producción
vercel rollback <DEPLOYMENT_URL>
```

Aprende este comando *antes* de necesitarlo, no durante la caída. Todo el sentido de la velocidad del vibe coding es que enviar cambios es barato — pero enviar barato solo se siente seguro cuando deshacer también es igual de barato.

Monitoreo y seguimiento de errores

Una vez que la gente usa tu app, necesitas saber cuándo se rompe — idealmente antes de que te lo digan ellos. El kit de inicio ligero:

- **Seguimiento de errores:** Sentry captura y agrupa excepciones con trazas de pila (stack traces), y te dice qué usuario y qué línea disparó cada una. Se instala con unas pocas líneas.
- **Monitoreo de disponibilidad:** un ping gratuito (UptimeRobot, Better Stack) visita tu sitio cada pocos minutos y te avisa por correo cuando cae.
- **Logs:** sabe dónde los guarda tu host (wrangler tail, el panel de Vercel, etc.) para poder investigar cuando algo se ve mal.

Pídele a la IA que agregue Sentry y conecte un endpoint básico de verificación de salud — una pequeña ruta como /health que devuelve 200 OK y confirma que la base de datos está accesible, la cual tu monitor de disponibilidad puede pingear. El objetivo no es una observabilidad de nivel empresarial — es simplemente *no enterarte de una caída por un tuit enojado*.

Rendimiento y caché

Las apps lentas pierden usuarios y rankean peor en las búsquedas. No necesitas microoptimizar el primer día, pero atrapa las victorias baratas:

- Imagina guardar copias de tus archivos en almacenes por todo el mundo para que a cada visitante lo atienda el más cercano a él. Eso es una **CDN** (Content Delivery Network, red de distribución de contenido) — sirve los recursos estáticos a través de una (la mayoría de los hosts lo hacen automáticamente).
- **Cachea** las respuestas y consultas de base de datos costosas que no cambian a menudo.
- Comprime y carga las imágenes de forma diferida (lazy-load); envía menos JavaScript.

- Ejecuta **Lighthouse** en Chrome DevTools y pásale el informe a la IA para que priorice las correcciones.

Pregunta: “Aquí está mi informe de Lighthouse. Dame los tres cambios con mayor impacto y menor esfuerzo.” Deja que la medición, no la intuición, guíe lo que optimizas — es genuinamente común pasar un día optimizando lo equivocado porque se *sentía* lento, cuando el costo real era una sola consulta sin índice o una imagen de portada de 4 MB que olvidaste comprimir.

Lista de verificación previa al lanzamiento

Antes de anunciar nada, recorre esta lista. Deja que la IA te ayude a verificar cada elemento en lugar de confiar en tu memoria.

- ☐ La app despliega sin errores y la URL de producción carga
- ☐ El dominio personalizado resuelve con HTTPS válido
- ☐ Todos los secretos están configurados en producción; ninguno subido al repositorio
- ☐ `.env` está en gitignore y `.env.example` está actualizado
- ☐ Los pagos se probaron de extremo a extremo (incluyendo el webhook) en modo de prueba
- ☐ El registro, el inicio de sesión y el flujo principal funcionan en producción
- ☐ El seguimiento de errores y el monitoreo de disponibilidad están activos
- ☐ Sabes cómo revertir, y lo has probado una vez
- ☐ El diseño móvil se revisó en un teléfono real
- ☐ Existe una política de privacidad y una página de términos básicas
- ☐ Las copias de seguridad de la base de datos están activadas
- ☐ Hay una forma clara de que los usuarios te contacten (correo o formulario)

Después de lanzar

Publicar es el comienzo, no el final. La primera versión es una hipótesis; los usuarios reales son el experimento. Observa tu panel de errores, lee cada mensaje, y envía correcciones pequeñas rápido. Resiste el impulso de reconstruir todo ante el primer indicio de fricción — repara los bordes filosos, redobla la apuesta en lo que la gente realmente usa, y deja que el resto espere.

El hábito más poderoso aquí es un bucle ajustado: notas un problema o una idea, se lo describes a la IA, envías el cambio, observas el resultado. Como el trabajo mecánico está delegado, puedes iterar en horas en lugar de semanas. Esa velocidad — no ninguna característica en particular — es la verdadera ventaja de construir con *vibe coding*. Estás en vivo. Ahora sigue adelante.

Repaso y práctica

Puntos clave

- Publicar es un flujo de trabajo, no un momento: desplegar, observar, corregir, repetir — y haz que revertir sea algo que ya has probado una vez.
- Ejecuta la lista de verificación previa al lanzamiento antes de anunciar nada, y deja que la IA verifique cada elemento en lugar de confiar en tu memoria.
- Deja que la medición guíe la optimización — un informe de Lighthouse le gana a una corazonada sobre qué “se siente lento” siempre.
- Los secretos viven en la configuración de producción, nunca en el repositorio; `.env` está en gitignore y `.env.example` se mantiene actualizado.
- La primera versión es una hipótesis; los usuarios reales son el experimento, así que itera con correcciones pequeñas y rápidas.

Ponlo en práctica

Toma una app que hayas desplegado (o despliega una pequeña ahora mismo) y recorre la lista de verificación previa al lanzamiento completa contra ella, línea por línea. Por cada casilla que no puedas marcar honestamente, pégale el elemento fallido a la IA y pide el cambio más pequeño que cierre la brecha. Termina haciendo clic en un enlace profundo del sitio en vivo — no solo la página de inicio — para confirmar que producción se comporta como en desarrollo.

Prompt del capítulo

```
Here is my deployed app and its stack: [describe app + hosting + services].  
Act as my launch reviewer. Walk this pre-launch checklist item by item  
and, for each, tell me how to verify it on the LIVE site (not in dev):  
- production URL loads over valid HTTPS on the custom domain  
- all secrets set in production; none committed to the repo  
- core happy path (sign-up, login, main action) works in production  
- payments tested end-to-end including the webhook (if any)  
- error tracking and uptime monitoring are live  
- I can roll back, and have tested it once  
- mobile layout works on a real phone  
- database backups are enabled  
For every item that fails, give me the SMALLEST change that fixes it.  
Do not assume anything passes — make me prove each one.
```

Casos de estudio: de cero al lanzamiento



La teoría es útil, pero nada enseña la programación asistida por IA (vibe coding) como verla suceder. Este capítulo recorre cuatro proyectos pequeños pero reales, contruidos y lanzados de principio a fin dirigiendo una IA. Ninguno es un juguete. Cada uno tiene usuarios, un nombre de dominio y dinero o tiempo en juego.

En cada caso verás la idea, la especificación que le dimos a la IA, la pila tecnológica que elegimos, los prompts que realmente marcaron la diferencia, los obstáculos que casi nos hicieron descarrilar y cómo salió a producción. Lee esto como manuales de procedimiento, no como una selección de momentos destacados. Lo importante no es que estos proyectos exactos importen, sino que los *movimientos* se repiten. Para el cuarto empezarás a reconocer el ritmo: una especificación corta, una pila gestionada, depuración basada en evidencia y un lanzamiento que se pone a prueba contra la realidad.

Caso de estudio 1: Una página de aterrizaje multilingüe con pagos

La idea

Una amiga vende una guía en PDF por \$29 para personas extranjeras que se mudan a Corea. Quería una sola página que cargara rápido en todo el mundo, hablara inglés, coreano y chino, y aceptara pagos con tarjeta sin que ella tuviera que tocar un servidor.

La especificación

Mantuvimos la especificación en un solo párrafo antes de escribir código:

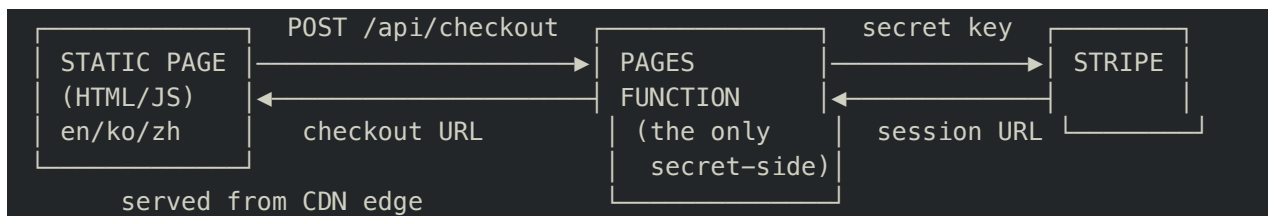
```
Build a single-page marketing site for a $29 digital product.
Three languages (en/ko/zh), auto-detected from the browser but
switchable with a toggle. One "Buy" button that opens Stripe
Checkout and redirects to a thank-you page on success. No backend
server I have to maintain. Must load fast from anywhere.
```

Fíjate en lo que la especificación *no* dice: ningún framework, ningún sistema de diseño, ninguna analítica, ninguna captura de correo electrónico. Cada una de esas habría sido una función razonable, y cada una habría retrasado el primer lanzamiento. La disciplina de una especificación de un párrafo es, en gran medida, la disciplina de dejar cosas fuera.

La pila tecnológica

“Sin backend que yo mantenga” más “rápido en todo el mundo” apuntaba directo a alojamiento estático en una CDN de borde (Content Delivery Network — red de distribución de contenido: copias de tus archivos guardadas en centros de datos de todo el mundo para que cada visitante cargue desde uno cercano) con una pequeña función sin servidor para lo único que *necesita* un secreto: crear la sesión de Stripe. Elegimos un sitio estático en Cloudflare Pages con una sola Pages Function. Toda la arquitectura cabe en una frase: una página estática que llama a una función que habla con Stripe, y esa simplicidad fue lo que la hizo depurable después.

Todo el sistema son tres cajas, y solo la del medio toca un secreto:



Los prompts clave

Empezamos en general y dejamos que la IA propusiera la estructura:

```
Scaffold a static landing page (plain HTML/CSS/JS, no framework)
with a language toggle for en/ko/zh. Store the copy in a single
JS object keyed by language. Detect the default from
navigator.language. Keep it to one index.html plus one main.js.
```

Luego el camino de pago, donde vive el secreto:

```
Add a Cloudflare Pages Function at /api/checkout that creates a
Stripe Checkout session for a single $29 product and returns the
redirect URL. Read STRIPE_SECRET_KEY from the environment, never
hardcode it. The Buy button should POST to this function and then
window.location to the returned URL.
```

La frase “never hardcode it” (nunca lo codifiques directamente) se gana su lugar. Si se la deja a sus propios valores por defecto, una IA felizmente coloca una clave de marcador de posición en línea para que el ejemplo “funcione”, y un marcador de posición tiene la costumbre de convertirse en una clave real que termina en producción. Decirlo una vez en voz alta, en el prompt, es más barato que atraparlo en la revisión.

El obstáculo

La primera prueba en vivo falló: el botón de comprar no hacía nada, y la consola del navegador mostraba un error de CORS (Cross-Origin Resource Sharing — el intercambio de recursos de origen cruzado: la regla

del navegador sobre qué sitios pueden llamar a qué otros sitios). No adivinamos. Pegamos el error exacto de vuelta:

```
Clicking Buy logs: "Access to fetch at '/api/checkout' blocked
by CORS policy." The function and page are on the same domain.
Here's the function code: [pasted]. What's actually wrong?
```

La IA lo detectó al instante: la función estaba devolviendo la URL de Stripe como JSON (JavaScript Object Notation — un formato de texto simple y legible que usan los programas para intercambiar datos estructurados), pero el fetch se estaba haciendo antes de que se manejara el preflight OPTIONS de la función, porque habíamos desplegado por error la página estática y la función en dos proyectos de Pages distintos. La solución real era de topología de despliegue, no de código. Movimos la función al directorio /functions del mismo proyecto y el error desapareció.

Vale la pena detenerse en la lección: el *síntoma* estaba en el código (un error de CORS proveniente de un fetch), pero la *causa* estaba en la infraestructura (dos proyectos en lugar de uno). Si hubiéramos dejado que la IA “arreglara el error de CORS” sin pegar la configuración, habría añadido cabeceras Access-Control-Allow-Origin como parche: una solución que compila, disimula el síntoma y deja viva la causa real del error. Pega el error literal y el contexto que lo rodea, y no aceptes la primera explicación plausible si no coincide con tu configuración.

El lanzamiento

Añadimos las claves de Stripe reales como variables de entorno cifradas en el panel de Pages (nunca en el repositorio), apuntamos el DNS (Domain Name System, el directorio de direcciones de internet que mapea un nombre a un servidor) de su dominio hacia Pages, e hicimos una compra de prueba real de \$29 con una tarjeta, que luego reembolsamos. También recorrimos a propósito el camino de fallo, con una tarjeta de prueba rechazada, para confirmar que no se le cobraba y que veía un mensaje coherente en vez de una pantalla en blanco. En marcha en una tarde. Hizo su primera venta esa misma noche.

Caso de estudio 2: Una pequeña herramienta SaaS con autenticación y base de datos

La idea

Una diseñadora independiente quería un panel privado para registrar horas facturables por cliente y exportar un CSV mensual (Comma-Separated Values — valores separados por comas: un archivo de hoja de cálculo en texto plano que cualquier aplicación de hojas de cálculo puede abrir). Nada elaborado, pero necesitaba cuentas (para que sus datos fueran solo suyos) y persistencia.

La especificación

La autenticación y una base de datos elevan la apuesta, así que la especificación se volvió más específica sobre los *límites*:

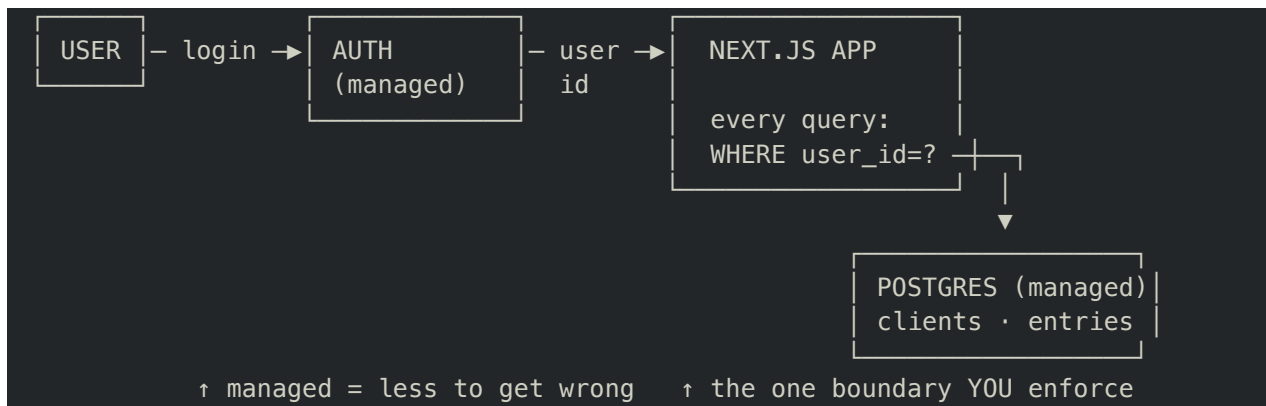
```
A logged-in web app where a user can: sign up / log in with email,
create clients, log time entries (date, client, hours, note),
see a table of entries filtered by month, and download that month
as CSV. Each user only ever sees their own data. Mobile-friendly.
```

“Each user only ever sees their own data” (cada usuario solo ve sus propios datos) parece una línea de experiencia de usuario. En realidad es el modelo de seguridad de toda la aplicación, comprimido en nueve palabras. Nombrarlo en la especificación nos permitió señalarlo cada vez que la IA se desviaba.

La pila tecnológica

Para alguien que construye en solitario, la jugada ganadora es una pila donde la autenticación y la base de datos son servicios gestionados, no código que tú escribes. Elegimos una app de Next.js desplegada en un host sin servidor, con una base de datos Postgres alojada y un proveedor de autenticación listo para usar que gestiona por nosotros el inicio de sesión por correo, las sesiones y los restablecimientos de contraseña. Menos código para equivocarse es menos código que una IA puede equivocarse en nuestro nombre. La autenticación en particular es una categoría que casi nunca quieres construir a mano con una IA: los modos de fallo son silenciosos, el radio de impacto abarca la cuenta de todos, y un proveedor gestionado ha tenido sus casos límite puestos a prueba por millones de inicios de sesión.

Las piezas gestionadas quedan por fuera; la única regla que tú controlas queda en el medio: cada consulta filtra por `user_id`, de modo que un usuario nunca puede leer las filas de otro:



Los prompts clave

Dejamos que la propia plantilla del proveedor de autenticación hiciera el trabajo pesado, y luego dirigimos a la IA para que capa por capa añadiera la lógica de dominio encima:

```
We're using [auth provider]'s Next.js starter. Add a Postgres schema with two tables: clients (id, user_id, name) and time_entries (id, user_id, client_id, date, hours, note). Every query MUST filter by the logged-in user's id from the session. Generate the migration and the typed data-access functions.
```

La línea “MUST filter by user_id” (debe filtrar por user_id) fue la frase más importante de todo el proyecto. Repetimos esa restricción en casi cada prompt que tocaba datos, porque el bug más aterrador de una app multiusuario es que un usuario vea las filas de otro. Repetirla se siente redundante mientras la escribes; es exactamente esa redundancia la que te salva, porque el modelo no tiene memoria de cuánto importa esa restricción entre prompts separados.

Para la exportación:

```
Add a /api/export route that takes a month (YYYY-MM), pulls the logged-in user's time_entries for that month joined to client names, and streams a CSV download. Reject the request if there's no valid session.
```

El obstáculo

Durante las pruebas, creamos dos cuentas y descubrimos que la cuenta B podía ver los clientes de la cuenta A en un menú desplegable. Este es exactamente el bug que temíamos. En lugar de pedirle a la IA que lo “arreglara”, primero le hicimos demostrar el problema:

```
Account B is seeing Account A's clients. Show me every database query in the codebase that reads the clients table, and for each one tell me whether it filters by the session user_id. Don't fix anything yet — just audit.
```

La auditoría reveló una consulta —la que cargaba el desplegable— que se había escrito antes de que añadiéramos la restricción y se había colado. Le pedimos que agregara el filtro que faltaba, y luego pedimos una salvaguarda:

```
Add a single helper that every read goes through, which takes the session and injects the user_id filter, so no future query can forget it. Refactor the existing queries to use it.
```

Eso convirtió una corrección puntual en una garantía estructural. Después hicimos que esa garantía fuera comprobable, porque una salvaguarda que no puedes verificar es solo una esperanza:

```
Write a test that creates two users, has each create a client, then asserts that user A's session can never read user B's client through any of the data-access functions.
```

La lección: cuando una IA introduce un fallo de seguridad, no te limites a parchar el caso puntual; dirígela para que elimine la *categoría* del error, y luego cierra esa categoría con llave mediante una prueba que falle ruidosamente si alguien vuelve a abrirla.

El lanzamiento

Sembramos un mes de datos de prueba, exportamos el CSV, lo abrimos en una hoja de cálculo para confirmar que los números y la codificación de caracteres eran correctos, y luego pusimos una contraseña fuerte a la base de datos y rotamos las credenciales fuera de cualquier archivo local. Desplegamos en el host sin servidor, añadimos las variables de entorno de producción en su panel, y le dimos la URL. Ella misma se registró con un alta real. Toda la construcción tomó un fin de semana.

Caso de estudio 3: Una utilidad de automatización personal

La idea

Uno puramente egoísta: un script que cada mañana toma los eventos del calendario del día y el clima, arma un resumen de una línea y lo envía a un chat personal de Telegram. El objetivo era dejar de abrir tres aplicaciones antes del café.

La especificación

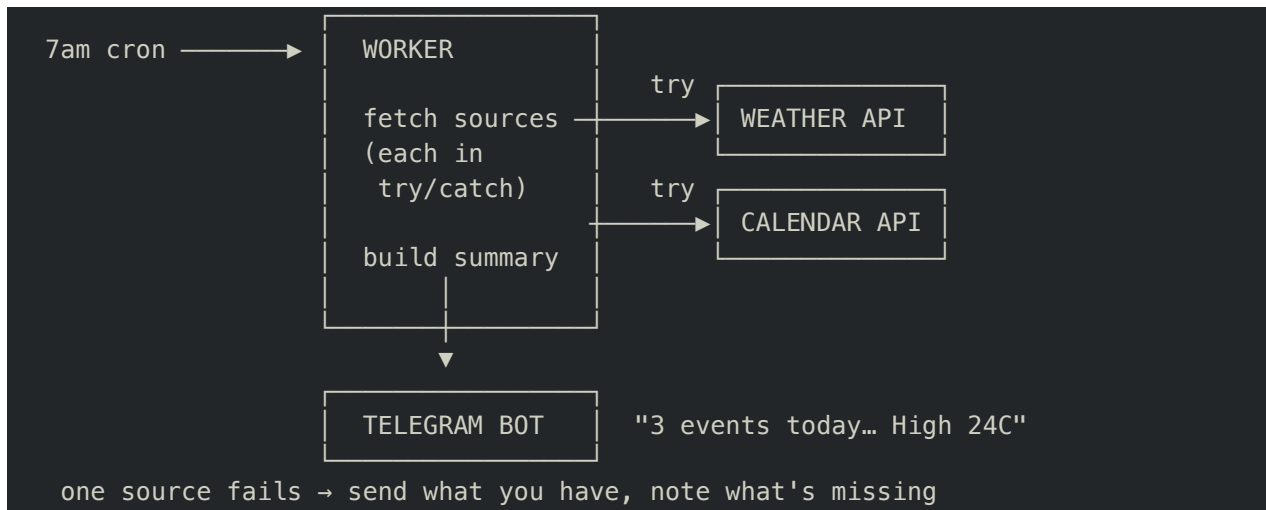
```
A scheduled job that runs at 7am my timezone. It reads today's events from my calendar, fetches the forecast for my city, and sends one Telegram message like: "3 events today, first at 9:30. High 24C, rain after 4pm." If any source fails, still send what you have and note what's missing.
```

Esa última frase —degradarse con elegancia en lugar de fallar por completo— es el tipo de comportamiento que hay que pedir de forma explícita. La IA no lo va a asumir. Si no se dice, el comportamiento por defecto al que un modelo recurre es “lanzar un error”, lo que para un trabajo cron de las 7 a. m. significa una mañana silenciosa y ninguna pista de por qué.

La pila tecnológica

Una sola función programada sin servidor (un Worker activado por cron) para que no haya ninguna máquina que mantener en funcionamiento. Todo el proyecto es un archivo más un horario. Para una utilidad personal esto importa más de lo que parece: cualquier cosa con un servidor que cuidar es algo que eventualmente olvidarás pagar, y el proyecto muere de abandono en lugar de fallo.

Un disparador se ramifica hacia dos fuentes y converge en un solo mensaje. El try/catch alrededor de cada fuente es lo que permite que un solo fallo se degrade con elegancia en lugar de matar toda la ejecución:



Los prompts clave

```
Write a Cloudflare Worker triggered by cron at 7am Asia/Seoul.
It calls a weather API and a calendar API (I'll provide both
tokens as secrets), builds a one-line summary, and POSTs it to
the Telegram Bot API. Wrap each external call in try/catch so one
failure doesn't kill the message. Read every token from env.
```

Cuando quisimos iterar sobre la redacción sin esperar hasta las 7 a. m.:

```
Add a manual trigger: if the Worker is hit with a GET request and
a ?test=1 query param, run the same logic immediately and return
the message text in the response instead of sending it. Keep this
behind a secret token so only I can trigger it.
```

Ese disparador de prueba nos salvó de un ciclo de retroalimentación brutal: podíamos ver el resultado a demanda en vez de una vez al día. Es una jugada general que vale la pena robar: cada vez que tu código se ejecute según un horario, constrúyete una forma de ejecutarlo *ahora mismo*, a demanda, antes de comprometerte a esperar al reloj.

El obstáculo

El mensaje de Telegram llegaba, pero las horas estaban mal: los eventos aparecían en UTC, no en hora coreana. El clima estaba bien. Lo aislamos:

```
Calendar times are 9 hours off - showing UTC, not Asia/Seoul.
The weather time ("rain after 4pm") is correct. Here's how I
```

```
format both: [pasted]. Fix only the calendar formatting and tell me why the weather path was already right.
```

La IA explicó que la API del clima (Application Programming Interface — la puerta que un servicio expone para que otro programa obtenga sus datos) ya devolvía cadenas localizadas, mientras que la API del calendario devolvía marcas de tiempo en UTC que estábamos imprimiendo tal cual. Añadió una sola conversión de zona horaria en el paso de formateo del calendario. Acotar la corrección a “solo el camino del calendario” evitó que reescribiera el código del clima que ya funcionaba, una forma común en que las correcciones de la IA rompen cosas que estaban bien. La segunda mitad de ese prompt — “tell me why the weather path was already right” (dime por qué el camino del clima ya estaba bien)— cumplió una doble función: nos dio la causa raíz real en lugar de una suposición, y obligó a la IA a articular la diferencia para que su corrección no “armonizara” accidentalmente ambos caminos en uno solo, roto.

El lanzamiento

“Lanzar” aquí solo significó desplegar el Worker, configurar el horario cron y añadir los tokens de API como secretos cifrados. Golpeamos el endpoint `?test=1` un par de veces para confirmar que el mensaje se leía bien, y luego lo dejamos correr. Ha funcionado cada mañana desde entonces, y la línea de degradación elegante demostró su valor un mes después, cuando la API del clima tuvo una interrupción y el mensaje igual llegó, solo que sin el pronóstico.

Caso de estudio 4: Un sitio de contenido que no tienes que atender

La idea

Un aficionado quería convertir una carpeta de notas en Markdown —reseñas de restaurantes de años de viajes— en un sitio público con capacidad de búsqueda, sin ningún CMS (Content Management System) en el que iniciar sesión y sin factura mensual. Agregar una nota, hacer push, listo.

La especificación

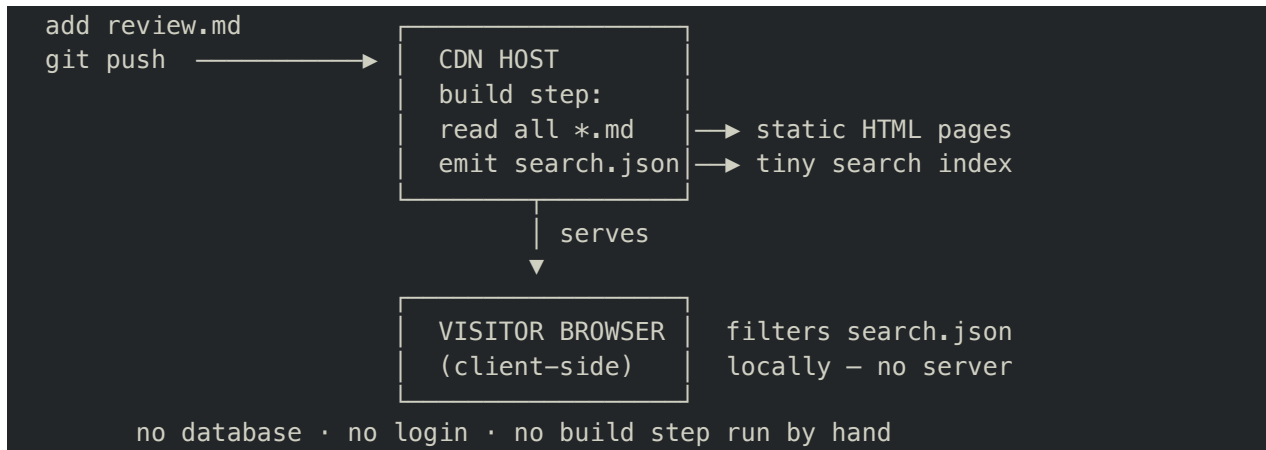
```
A static site generated from a folder of Markdown files, one file per review (title, city, rating, body in frontmatter). Build a homepage that lists reviews newest-first, a page per review, and client-side search by city or name. No database, no login, no build step I have to run by hand. Pushing a new .md file should publish it.
```

El requisito oculto está en la última frase. “No build step I have to run by hand” (ningún paso de construcción que tenga que ejecutar a mano) significa que la especificación en realidad trata del *flujo de trabajo*, no solo del resultado, y los requisitos de flujo de trabajo son de los que las IA se saltan a menos que los nombres, porque no aparecen en la aplicación en funcionamiento.

La pila tecnológica

Un generador de sitios estáticos que lee Markdown en tiempo de construcción, desplegado en un host de CDN con construcciones activadas por Git. Se hace push al repositorio, el host reconstruye, la CDN lo sirve. La búsqueda es del lado del cliente sobre un pequeño índice JSON generado en la construcción, así que no hay servidor ni costo de consulta: un buen intercambio con unos pocos cientos de reseñas, y uno que nombramos deliberadamente en lugar de descubrirlo por accidente.

Todo el flujo de trabajo es una sola flecha: un push activa una construcción que publica. No se ejecuta nada en el momento de la solicitud, salvo el propio navegador del visitante filtrando un índice:



Los prompts clave

Anclamos a la IA a la forma de los datos primero, antes de cualquier interfaz:

```
Set up a static site that reads every .md file in /reviews. Each file's frontmatter has title, city, rating (1-5), date. Parse all of them at build time into a sorted list (newest first) and generate: a homepage listing them, and one page per review at /reviews/<slug>. Show me the data-loading code first, before any styling.
```

“Before any styling” (antes de cualquier estilo) es una palanca deliberada. Pide todo de una vez y obtienes una página hermosa conectada a datos en los que no puedes confiar; pide primero la capa de datos y puedes verificar los cimientos antes de que un solo píxel te distraiga. Luego la búsqueda, acotada para mantenerla barata:

```
Generate a search.json at build time with {title, city, slug} for every review. On the homepage, add a search box that filters the visible list client-side by matching city or title – no network calls, just filter the already-loaded index. Keep it under 50 lines of JS.
```

El obstáculo

La construcción pasaba en local, pero el sitio desplegado devolvía 404 en cada página individual de reseña. La página principal funcionaba; las páginas de reseñas individuales no. Pegamos el síntoma y la configuración en lugar de teorizar:

```
Homepage works live, but /reviews/<slug> pages all 404 in production while working in local dev. Here's my build output directory listing and my host's deploy config: [pasted]. Why does local dev serve these pages but production doesn't?
```

```
The AI traced it to the difference between dev-server routing (which resolves paths dynamically) and static hosting (which serves only files that physically exist). The build was generating the review pages into the wrong output folder, so they never got uploaded – dev had hidden the bug because its router faked the routes that production couldn't.
```

Apuntamos el directorio de salida hacia lo que el host realmente desplegaba, las páginas aparecieron, y añadimos una línea a la lista de verificación de lanzamiento: *hacer clic en un enlace profundo en el sitio en vivo, no solo en la página principal*. La lección más amplia es que “funciona en desarrollo” y “funciona desplegado” son afirmaciones distintas, y la exportación estática es justo donde divergen: los servidores de desarrollo son indulgentes con las rutas de una forma en que el alojamiento real basado en archivos nunca lo es.

El lanzamiento

Publicamos una reseña real, vimos al host reconstruir, y cargamos un enlace profundo a esa reseña específica en el dominio en vivo, exactamente el caso sobre el que desarrollo había mentido. Luego buscamos su ciudad para confirmar que el índice la había recogido. Añadir la *siguiente* reseña fue un commit de una sola línea, que era todo el punto: el lanzamiento no fue un momento, fue un flujo de trabajo que sigue funcionando sin nosotros.

Qué tienen en común los cuatro

- ***La especificación vino primero, y fue corta.*** Un párrafo, no un documento. Pero nombró las restricciones que importaban: idiomas, “solo ve sus propios datos”, “aun así envía lo que tengas”, “publicar al hacer push de un archivo”, incluyendo las restricciones de flujo de trabajo que nunca aparecen en la aplicación en funcionamiento.
- ***Los secretos nunca tocaron el repositorio.*** Cada proyecto leyó las claves desde variables de entorno o secretos cifrados, configurados en un panel, no en el código. Dijimos “never hardcode it” en voz alta, porque el comportamiento por defecto es el contrario.
- ***La pila se eligió para que hubiera menos margen de error.*** Autenticación gestionada, base de datos gestionada, alojamiento estático, funciones sin servidor: cada elección eliminó una categoría de código que la IA podía romper en nuestro nombre, y una categoría de cosa que tendríamos que mantener.
- ***Los obstáculos se resolvieron alimentando a la IA con evidencia exacta*** — el error literal, el código real, la configuración de despliegue, el síntoma verdadero— y acotando las correcciones de forma estrecha para que el código que funcionaba siguiera funcionando. Dos veces el síntoma estuvo en el código, pero la causa estuvo en la infraestructura.
- ***El lanzamiento significó una prueba real contra la realidad:*** una compra en vivo (y una rechazada), una segunda cuenta de usuario, un disparador a demanda, un enlace profundo en el sitio desplegado. No “compiló”.

No necesitas una gran idea para empezar. Necesitas una pequeña, una especificación de un párrafo, y la disposición a dirigir en lugar de escribir, además de la paciencia para poner a prueba lo construido contra la realidad antes de darlo por terminado.

Recapitulación y práctica

Puntos clave

- La especificación vino primero y fue corta: un párrafo que nombró las restricciones que importaban, incluyendo las restricciones de flujo de trabajo que nunca aparecen en la aplicación en funcionamiento.
- Los secretos nunca tocaron el repositorio; cada proyecto leyó las claves desde variables de entorno o configuración cifrada establecida en un panel.
- La pila se eligió para que hubiera menos margen de error: autenticación gestionada, base de datos gestionada, alojamiento estático, funciones sin servidor.
- Los obstáculos se resolvieron alimentando a la IA con evidencia exacta — el error literal, el código real, la configuración de despliegue— y acotando las correcciones de forma estrecha.
- El lanzamiento significó una prueba real contra la realidad, no “compiló”: una compra en vivo, una segunda cuenta, un enlace profundo en el sitio desplegado.

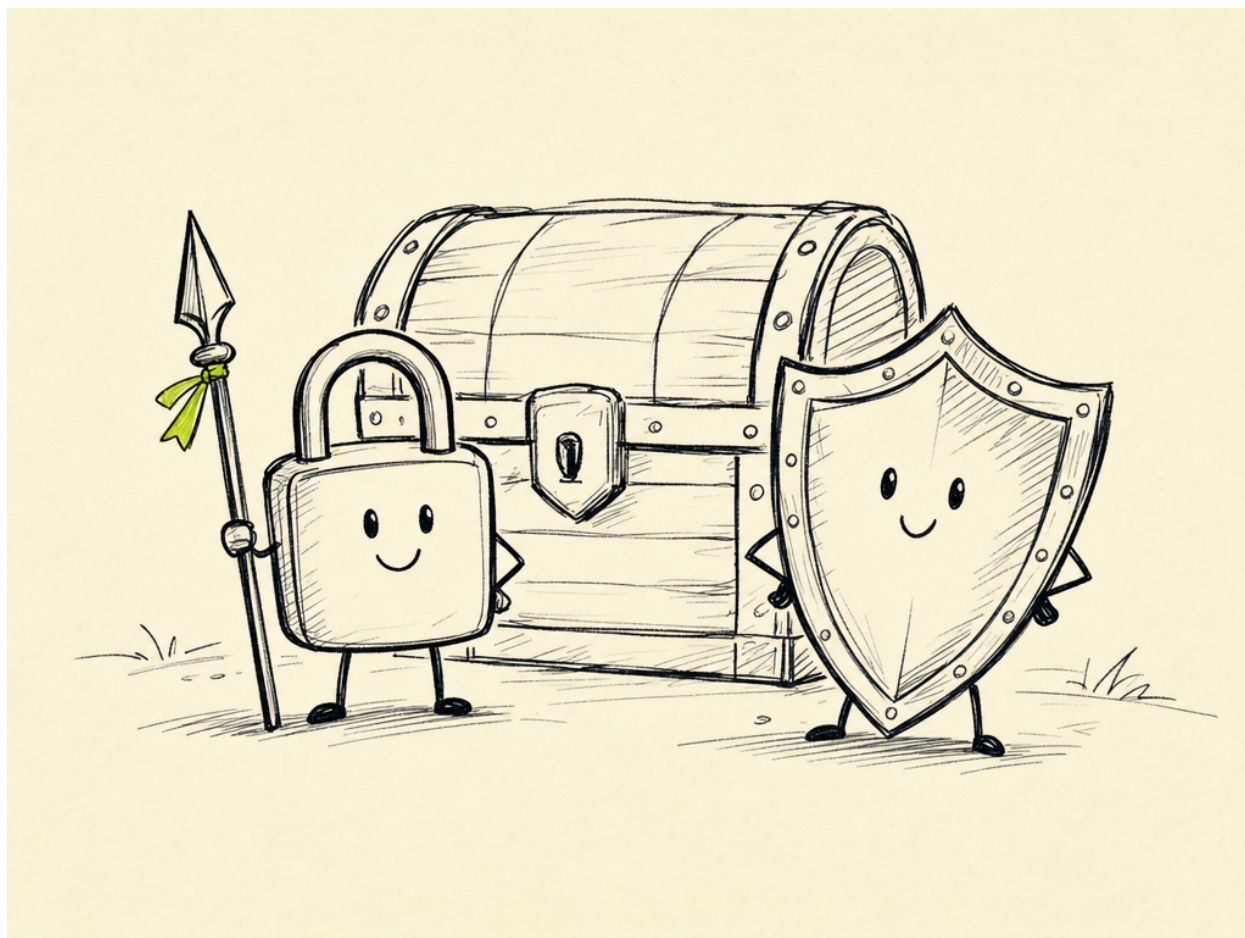
Ponlo en práctica

Elige una de tus propias ideas y escribe su especificación de un párrafo tal como lo hicieron estos cuatro casos: nombra la acción central, los datos que cada usuario puede ver, y al menos una restricción de flujo de trabajo (cómo se publica el contenido, quién puede hacer qué). Luego enumera los bloques de construcción gestionados en los que te apoyarías para que haya menos código propio que pueda romperse. Mantenlo en un solo párrafo más una breve lista de puntos; si crece más allá de eso, estás sobre-especificando.

Prompt del capítulo

```
I want to build: [one-paragraph description of the app].
Before any code, help me tighten this into a buildable spec.
1. Restate the single core action a user performs.
2. List the data each user can see – and what they must NOT see.
3. Name the workflow constraints that won't show up in the UI
   (e.g. "pushing a file should publish it", "only their own data").
4. Recommend a stack that MINIMIZES custom code: managed auth,
   managed database, static or serverless hosting where possible.
5. Flag the one part most likely to go wrong, and how we'll test it
   against reality (a real purchase, a second account, a deep link).
Keep the spec to a paragraph plus bullets. Push back if it's too big.
```

Seguridad para vibe coders



La seguridad es la parte del vibe coding que muere en silencio. Una función rota se nota de inmediato: el botón no funciona, lo arreglas. Un agujero de seguridad no se nota nunca, hasta el día en que alguien lo encuentra y publica los datos de tus usuarios en un foro. Para entonces, la IA que escribió el fallo ya se ha ido hace tiempo, y eres tú quien tiene que explicarle a mil personas por qué se filtró su contraseña.

Este capítulo no trata de meter miedo. No necesitas convertirte en un pentester (probador de penetración, alguien que ataca sistemas de forma controlada para encontrar fallos) ni memorizar el OWASP Top 10 para lanzar una pequeña app segura. Necesitas entender un puñado de formas en que el software se ve comprometido, reconocer los patrones que la IA repite una y otra vez, y añadir un filtro de revisión antes de lanzar. La filosofía es la misma que en el resto de este libro: **nunca lances algo que no entiendes** — y la seguridad es el terreno donde no entender sale más caro.

Por qué el código generado por IA es inseguro por defecto

La IA optimiza para el prompt que le diste, y tu prompt casi siempre dice “que funcione”, nunca “que sea seguro”. Así que produce el camino más corto hacia una función que funciona. Ese camino más corto suele ser el inseguro: consultas construidas por concatenación de cadenas de texto, un endpoint (una de las puertas direccionables de tu app — una URL concreta donde la app responde a peticiones) sin verificación de permisos, un secreto pegado directamente en el código porque así era el ejemplo del que aprendió.

Y va a peor, porque la IA aprendió de todo internet, incluida una década de tutoriales y repositorios abandonados que ya eran inseguros de por sí. El código inseguro es *más* frecuente en sus datos de entrenamiento que el código seguro, porque el código seguro es más difícil de escribir y más raro de publicar. Cuando

pides “un formulario de inicio de sesión”, lo más probable estadísticamente es que obtengas el formulario de inicio de sesión promedio de internet, y el formulario de inicio de sesión promedio de internet tiene problemas.

La IA tampoco tiene ninguna noción de amenaza. Una persona que construye una función de subida de archivos siente un destello de preocupación: *¿y si alguien sube algo malicioso?* La IA no siente nada. Escribe el manejador con la misma confianza tanto si es hermético como si está abierto de par en par. No hay vacilación, ni un “hmm, esta parte me pone nervioso”. La cuota de nerviosismo tienes que ponerla tú.

La conclusión no es “el código de la IA es peligroso, no lo uses”. Es que **“funciona” y “es seguro” son dos preguntas distintas**, y la IA solo responde a la primera a menos que la obligues a responder también a la segunda.

Inyección: cuando la entrada se convierte en código

La inyección es el fallo grave más antiguo y todavía el más común. Ocurre cuando datos proporcionados por un usuario se tratan como *instrucciones* en lugar de como *datos*. Dos variantes importan para la mayoría de las apps.

Inyección SQL — SQL (Structured Query Language, el lenguaje que las apps usan para pedirle datos a una base de datos) — ocurre cuando la entrada del usuario se cuela dentro de una consulta a la base de datos. El patrón vulnerable clásico:

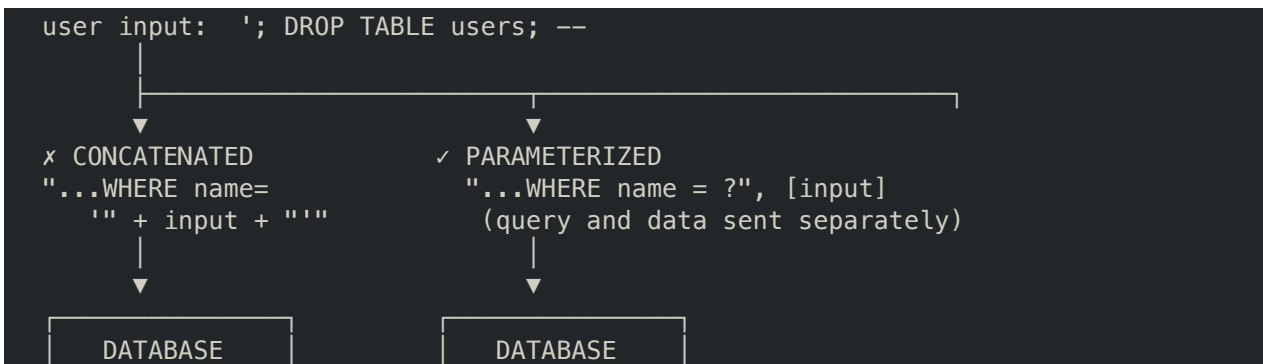
```
// VULNERABLE: the user's input is glued straight into the query
app.get("/user", (req, res) => {
  const name = req.query.name;
  db.query(`SELECT * FROM users WHERE name = '${name}'`);
});
// If someone passes name = '; DROP TABLE users; --
// your query becomes a command to delete the table.
```

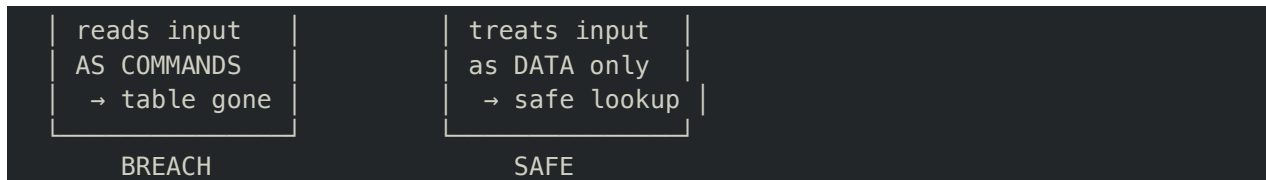
La solución es una **consulta parametrizada** (también llamada sentencia preparada). Envías la consulta y los datos por separado, de modo que la base de datos nunca confunde uno con otro:

```
// SAFE: the value is passed as a parameter, never as code
app.get("/user", (req, res) => {
  const name = req.query.name;
  db.query("SELECT * FROM users WHERE name = ?", [name]);
});
```

La regla: **nunca construyas una consulta por concatenación de cadenas de texto**. Si ves comillas invertidas o + ensamblando SQL alrededor de una variable en un diff, detente y pide la versión parametrizada.

Los dos caminos meten la misma entrada del usuario en la base de datos, pero solo uno evita que la base de datos confunda esa entrada con comandos:





XSS (cross-site scripting, o secuencias de comandos entre sitios) es la misma idea pero en el navegador. Si tomas la entrada de un usuario y la vuelcas en una página como HTML crudo (HyperText Markup Language, el código que le indica al navegador qué mostrar), un atacante puede inyectar una etiqueta `<script>` que se ejecuta en los navegadores de tus otros usuarios, robando sus sesiones, por ejemplo. La solución es el escapado: renderizar el contenido del usuario como *texto*, no como HTML. Los frameworks modernos como React escapan por defecto, lo cual es genial, hasta que la IA recurre a una vía de escape como `dangerouslySetInnerHTML` o `innerHTML` para “hacer que funcione”. Esas rutas evitan la protección. Trata cualquier llamada de ese tipo en un diff como algo que hay que cuestionar.

El principio unificador detrás de todo fallo de inyección: **mantén los datos y el código separados**. Cada vez que la entrada de un usuario cruza hacia una consulta, un comando, una plantilla o HTML, algo debe escaparla o parametrizarla.

Secretos y claves de API: la trampa de la clave expuesta

Un secreto es cualquier cosa que otorga acceso: claves de API (Application Programming Interface, Interfaz de Programación de Aplicaciones), contraseñas de bases de datos, tokens de proveedores de pago, claves de firma. El error de vibe coding más común — y más caro — es filtrar uno.

Dos trampas atrapan a la gente constantemente:

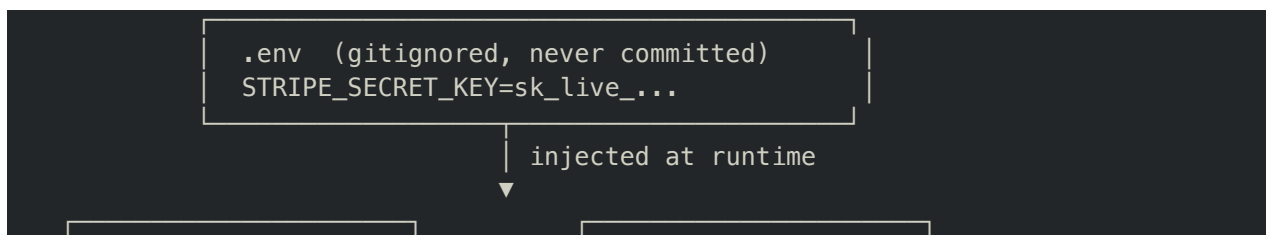
- **Secretos en el código del cliente.** Todo lo que está en tu frontend (el JavaScript que se ejecuta en el navegador) es *público*. Los usuarios pueden abrir las herramientas de desarrollador y leerlo. La IA, cuando se le pide que llame a una API desde un componente de React, pegará alegremente tu clave secreta ahí mismo — y ahora cualquiera que visite tu sitio puede copiarla y disparar tu factura. Las claves secretas pertenecen al servidor, nunca al código que se envía al navegador.
- **Secretos en el repositorio.** Una clave escrita directamente en un archivo se termina subiendo a git. Aunque la borres más tarde, sigue viva para siempre en el historial de git, y hay bots que escanean repositorios públicos buscando justo esto a los pocos minutos de un push.

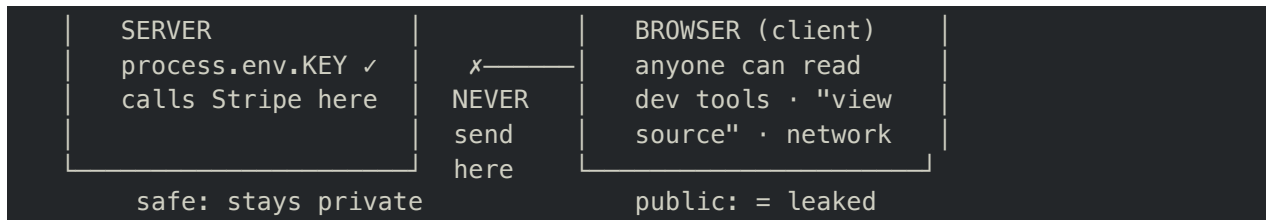
El lugar correcto para un secreto es una **variable de entorno** — un valor que se le suministra a la app en tiempo de ejecución, mantenido completamente fuera del código:

```
// VULNERABLE: key is in the source, will be committed to git
const stripe = new Stripe("sk_live_51H8xQ2eZvK...");

// SAFE: key is read from the environment, never written in code
const stripe = new Stripe(process.env.STRIPE_SECRET_KEY);
```

Un secreto tiene exactamente un hogar seguro: del lado del servidor, leído desde el entorno. El navegador es un lugar público: cualquier cosa enviada ahí puede leerla cualquiera que abra las herramientas de desarrollador.





Añade `.env` (y `.env.local`, etc.) a tu `.gitignore` *antes* de escribir un solo secreto. Si alguna vez una clave termina en tu código o en tu historial, trátala como quemada: róta la (genera una nueva y revoca la vieja) — borrar la línea no basta, porque la clave vieja sigue siendo válida y sigue estando ahí fuera.

Autenticación vs. autorización: el endpoint que cualquiera puede llamar

Estas dos palabras suenan parecido, y la diferencia entre ellas es donde viven las brechas reales.

- **Autenticación** es *¿quién eres?* — iniciar sesión, demostrar identidad.
- **Autorización** es *¿qué se te permite hacer?* — si este usuario ya identificado puede realizar esta acción sobre estos datos.

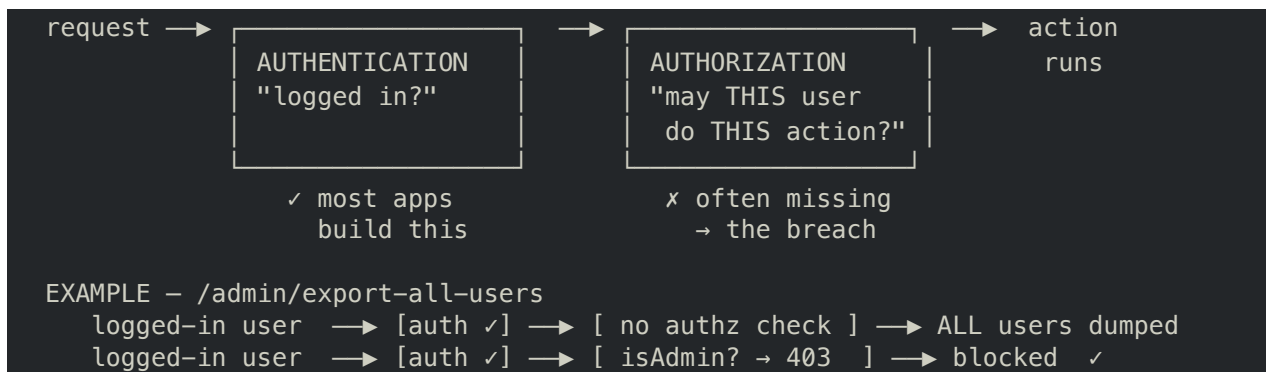
La IA es decente en autenticación; las bibliotecas se encargan de la mayor parte. La autorización es donde falla sistemáticamente, porque la autorización es específica de las reglas de tu app, y la IA no las conoce. El desastre de manual:

```
// VULNERABLE: checks you're logged in, but not WHO you are
app.get("/admin/export-all-users", requireLogin, (req, res) => {
  res.json(db.getAllUsers()); // any logged-in user can hit this
});
```

Ese endpoint está “protegido”: tienes que haber iniciado sesión. Pero *cualquier* usuario con sesión iniciada, incluido uno que se registró hace treinta segundos, puede llamarlo y descargar los datos de todos los usuarios. Autenticación presente, autorización ausente. La solución es verificar el permiso sobre la acción en sí:

```
// SAFE: confirms this user is actually an admin
app.get("/admin/export-all-users", requireLogin, (req, res) => {
  if (!req.user.isAdmin) return res.status(403).send("Forbidden");
  res.json(db.getAllUsers());
});
```

Piensa en dos puertas que una petición debe atravesar. La autenticación pregunta “¿has iniciado sesión?”; la autorización pregunta “¿se te permite a *ti* hacer esto?” La brecha ocurre cuando una app construye la primera puerta y se olvida de la segunda:



La misma trampa aparece en miniatura por todas partes: un endpoint que devuelve el pedido #1234 sin

verificar que ese pedido pertenece al usuario que hace la petición. Cualquiera puede cambiar el número en la URL y leer el pedido de otra persona. La regla no tiene nada de glamurosa, pero es absoluta: **verifica la autorización en cada endpoint que toque datos, y no confíes en un ID que venga del cliente**. No asumas que una URL oculta es segura solo porque está oculta — “nadie sabe que esto existe” no es un control de seguridad.

Manejo de archivos y subidas

Dejar que los usuarios suban archivos es un campo minado silencioso, y el primer borrador de la IA casi nunca cierra los huecos:

- **Valida el tipo y el tamaño.** Sin límites, alguien sube un archivo de 10 GB o un ejecutable. Comprueba el contenido real, no solo la extensión del nombre de archivo, que cualquiera puede renombrar.
- **Nunca confíes en el nombre del archivo.** Un nombre como `../../etc/passwd` puede hacer que un manejador ingenuo escriba fuera de la carpeta prevista (esto se llama traversal de rutas). Genera tú mismo nombres seguros en lugar de usar el que te proporcionan.
- **No sirvas las subidas desde el mismo origen que tu app**, y no dejes que los archivos subidos se ejecuten como código. Un archivo suministrado por un usuario que cae en una carpeta que tu servidor va a *ejecutar* es un camino directo a la toma de control total.

No tienes que construir todo esto tú mismo — un servicio de almacenamiento se encarga de la mayor parte —, pero tienes que *pedirlo*, porque la IA no lo va a ofrecer por voluntad propia. “Añade una función de subida” te consigue el camino feliz. “Añade una función de subida que valide el tipo y el tamaño del archivo, rechace cualquier cosa que no sea una imagen, y almacene los archivos con nombres generados” te consigue algo más cercano a seguro.

Riesgo de dependencias: el paquete que la IA se inventó

Las apps modernas se apoyan en docenas de paquetes de terceros, y la IA los sugiere sin ningún reparo. Van de la mano dos riesgos.

Primero, el **typosquatting (registro malicioso de nombres casi idénticos) y los paquetes alucinados**. Los atacantes publican paquetes maliciosos con nombres a un pelo de los reales (requeusts en lugar de request), apostando a un error tipográfico. La IA a veces importa con toda confianza un paquete que no existe — y un atacante que note ese hábito puede registrar exactamente ese nombre con malware dentro. Antes de instalar cualquier cosa que la IA sugiera, échale un vistazo: ¿existe realmente, tiene cifras de descarga reales y un repositorio, está escrito el nombre como cabría esperar?

Segundo, las **dependencias no revisadas en general**. Cada paquete que añades es código que corre con el acceso completo de tu app. Más dependencias significa más superficie tanto para bugs como para ataques a la cadena de suministro. Prefiere unas pocas bibliotecas bien conocidas antes que una larga cola de otras oscuras, y pregúntate “¿de verdad necesitamos un paquete para esto, o son diez líneas?” antes de añadir uno.

Ejecuta periódicamente la herramienta de auditoría de tu ecosistema (`npm audit`, `pip-audit` y similares) — señala vulnerabilidades conocidas en lo que ya has instalado, y la IA es buena arreglando lo que reporta.

El filtro de revisión de seguridad

Aquí está el único hábito que convierte todo lo anterior de una preocupación en un proceso: antes de lanzar, **haz que la IA ataque su propio código**. El modelo que escribió la función normalmente puede encontrar los agujeros que tiene — solo que no lo hará a menos que se lo pidas. Cámbiale el rol de constructor a adversario:

You wrote this endpoint. Now act as an attacker trying to break it.
List every way a malicious user could:

- read or modify data they shouldn't (authorization holes)
- inject code via input (SQL injection, XSS, command injection)
- abuse missing validation or rate limits

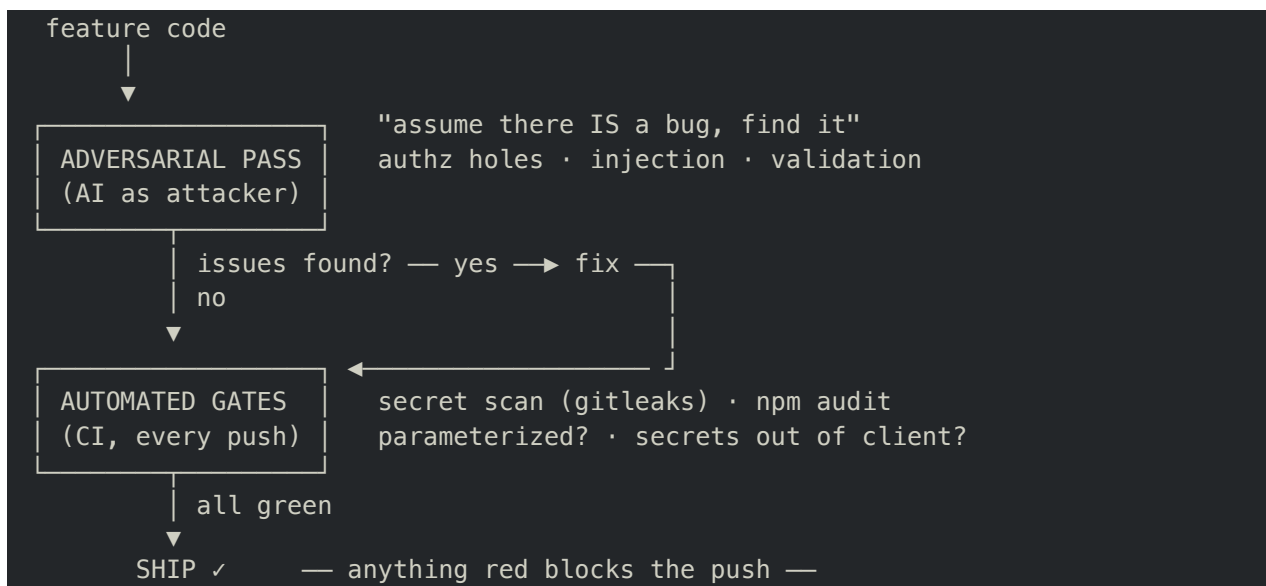
For each, show the exact request that exploits it, then the fix.

Don't reassure me – assume there IS a vulnerability and find it.

Esa última línea importa: si se le deja neutral, la IA tiende a decir “¡se ve seguro!”. Cuando se le dice que asuma que existe un fallo, de verdad se pone a buscarlo. Combina el pase adversarial con una breve lista de comprobación previa al lanzamiento que ejecutes sobre cualquier cosa de cara al usuario:

- ☐ Todo endpoint verifica la autorización, no solo que el usuario haya iniciado sesión
- ☐ Todas las consultas a la base de datos están parametrizadas — nada de SQL construido por cadenas de texto
- ☐ La entrada del usuario que se renderiza en la página está escapada (sin inyección de HTML crudo)
- ☐ No hay secretos en el código del cliente, y ninguno subido al repositorio
- ☐ .env está en el gitignore; cualquier clave filtrada ha sido rotada
- ☐ Las subidas de archivos validan tipo y tamaño y usan nombres generados
- ☐ Las nuevas dependencias fueron revisadas por su existencia real y su reputación

Imagina el filtro como una secuencia por la que nada se lanza sin pasar. La IA cambia de sombrero, de constructora a atacante, y luego los escaneos automatizados respaldan lo que un repaso humano pueda pasar por alto:



Y ejecuta un **escáner de secretos** antes de hacer push — una herramienta como gitleaks (o el escaneo integrado de tu plataforma) rastrea tu código y tu historial buscando cosas con forma de clave. Es una red de seguridad de un solo comando para el error más caro de la lista, y puedes hacer que la IA lo integre en la CI (Continuous Integration, Integración Continua — el robot que ejecuta automáticamente tus comprobaciones cada vez que subes código) para que se ejecute en cada push.

Complacencia con la automatización, edición seguridad

El momento más peligroso no es el día uno — es el día noventa. El día uno eres cauteloso; lees cada diff, ejecutas el prompt de atacante, revisas el manejador de subidas. Luego se lanzan cien cambios y no pasa nada malo, y las comprobaciones empiezan a sentirse como ceremonia. Te saltas el filtro de revisión “solo esta vez” en un cambio pequeño. Ese cambio pequeño añade un endpoint sin verificación de autorización. No pasa nada durante un tiempo, porque que no pase nada es exactamente el aspecto que tiene un agujero de seguridad desde dentro.

Esto es complacencia con la automatización aplicada a la seguridad, y es peor aquí que en cualquier otro sitio, porque los fallos de seguridad son silenciosos y diferidos. Una función rota te castiga en minutos. Una verificación de permisos rota te recompensa con éxito aparente hasta el mismísimo día de la brecha. La ausencia de desastre no es evidencia de seguridad — es la condición normal de una app vulnerable que simplemente aún no ha sido encontrada.

La defensa consiste en hacer que el filtro sea lo bastante barato como para no saltártelo y lo bastante automático como para que no puedas. Pon el escaneo de secretos y el `audit` en la CI para que se ejecuten sin depender de tu memoria, y mantén a mano el prompt de atacante para cualquier cosa que toque autenticación, datos o subidas. Mantén una línea firme: un cambio que maneje datos de usuario o permisos no se lanza hasta que alguien — tú, con la ayuda honesta de la IA — haya intentado activamente romperlo. La velocidad en el *vibe coding* viene de buenos filtros, no de confiar en que el silencio significa que todo va bien.

Resumen y práctica

Ideas clave

- El código generado por IA es inseguro por defecto; asume que existe una vulnerabilidad y ponte a buscarla en lugar de confiar en una primera impresión limpia.
- Las grandes categorías se repiten: inyección, secretos expuestos, autorización ausente, subidas de archivos inseguras y dependencias inventadas.
- La autorización es por petición, no por sesión iniciada — cada endpoint debe verificar que este usuario puede hacer esta acción, no solo que ha iniciado sesión.
- Ejecuta un escáner de secretos como `gitLeaks` antes de cada push, y rota cualquier clave que alguna vez se haya filtrado; integra el escaneo en la CI para que no puedas saltártelo.
- El día noventa es más peligroso que el día uno — haz que el filtro de seguridad sea lo bastante barato como para no saltártelo y lo bastante automático como para que no puedas.

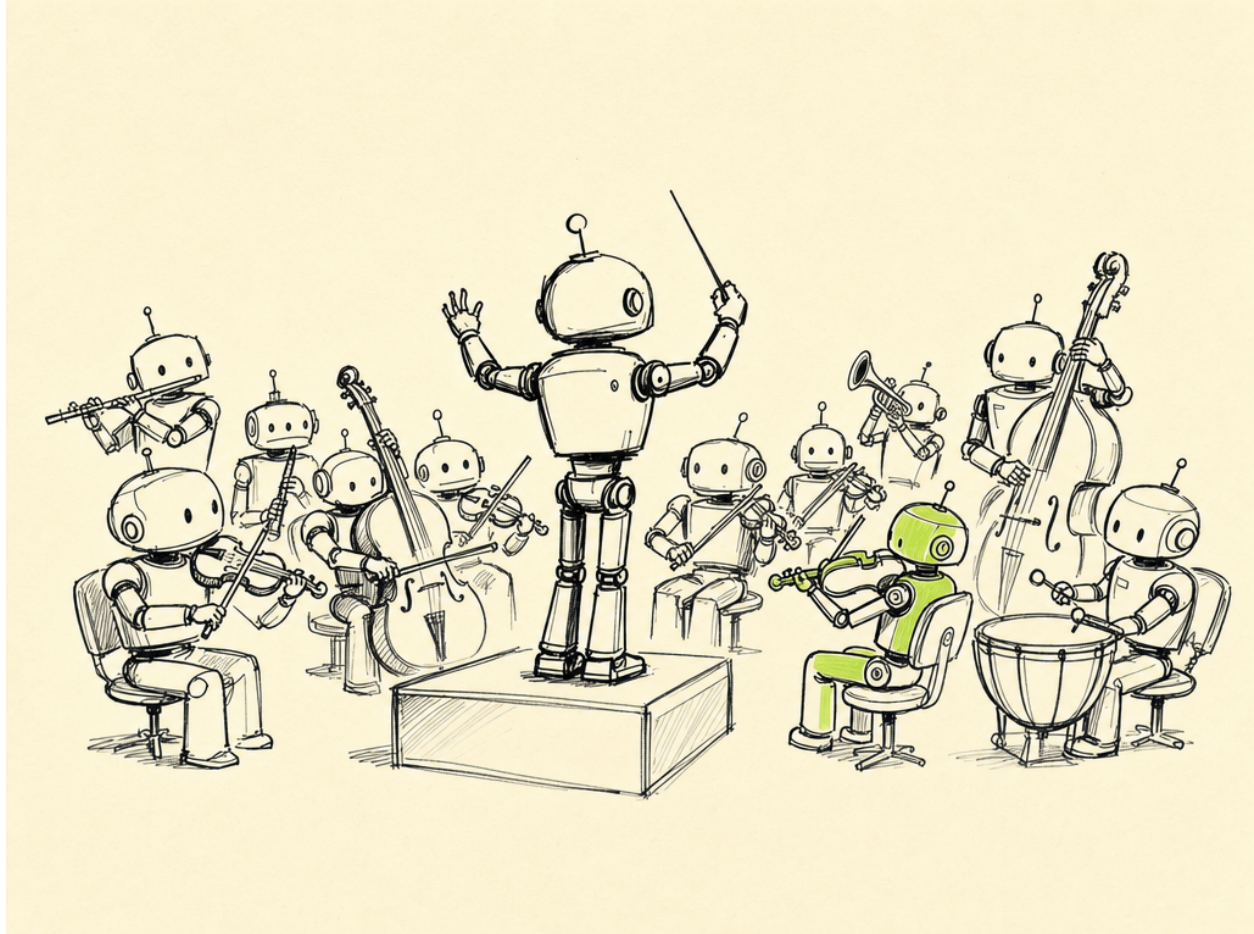
Pruébalo

Toma cualquier función tuya que toque autenticación, datos de usuario o subidas y ejecútale el prompt de revisión adversarial de abajo. Por cada vulnerabilidad que la IA reporte, pide la petición exacta que la explota y la solución mínima, luego aplica la solución y vuelve a ejecutar el prompt. Termina ejecutando `gitLeaks` (o el escaneo de secretos de tu plataforma) sobre el repositorio y su historial.

Prompt del capítulo

```
Here is the code for a feature that handles [auth / user data / uploads]:
[paste the code].
Act as an attacker reviewing this for security holes. Specifically check:
- broken authorization (can a user act on another user's data?)
- injection via input (SQL injection, XSS, command injection)
- exposed secrets or keys in code or responses
- unsafe file handling (type, size, path, generated names)
- missing validation or rate limits
For EACH issue, show the exact request that exploits it, then the fix.
Do NOT reassure me — assume there IS a vulnerability and find it.
```

Orquestación de agentes



Durante la mayor parte de este libro, la IA ha sido una herramienta que sostienes en la mano: preguntas, responde, revisas el resultado. Ese sigue siendo el modelo mental correcto la mayor parte del tiempo. Pero las herramientas han desarrollado un segundo modo — uno en el que entregas un *objetivo* en lugar de un *paso*, y la IA trabaja durante minutos (a veces más) antes de volver a ti.

Este capítulo trata sobre ese cambio, y sobre no perder el hilo cuando ocurre. La tentación es leer “agente” como “ya puedo dejar de prestar atención.” Esa es la lectura que mete a la gente en problemas. Un agente es apalancamiento sobre trabajo que ya entiendes — no una licencia para lanzar código que has dejado de mirar.

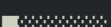
Los tres niveles

Ayuda ver la asistencia de IA como tres niveles, cada uno entregándote un poco más de cuerda.

- **Autocompletados en el editor.** La IA sugiere la siguiente línea o bloque mientras escribes. Tienes control total; aceptas o rechazas cada sugerencia. La unidad de trabajo son unas pocas líneas, y las revisas todas por defecto porque aparecen justo debajo de tu cursor.
- **Asistentes a nivel de repositorio.** Conversas con un asistente que puede ver todo tu proyecto, editar varios archivos a la vez, ejecutar comandos y leer la salida. Aquí es donde ocurre la mayor parte del trabajo de este libro. La unidad de trabajo es un paso o una pequeña funcionalidad, y revisas el diff después de cada turno.
- **Agentes autónomos.** Das un objetivo — “haz que pase la suite de pruebas,” “migra este módulo

a la nueva API (Application Programming Interface, interfaz de programación de aplicaciones — el conjunto de comandos que una pieza de software expone para que otras la usen)” — y el agente planifica, edita, ejecuta, lee errores y vuelve a intentarlo a través de muchos pasos. Revisas el *resultado*, no cada acción individual. La unidad de trabajo es una tarea completa.

Cada nivel te entrega más cuerda: una unidad de trabajo más grande, una dirección menos continua, y un punto de revisión más tardío:

TIER	UNIT OF WORK	YOU REVIEW	CONTROL
completions ▼	a few lines	each suggestion	 tight
repo assistant ▼	a step/feature	the diff/turn	
autonomous agent	a whole task	the RESULT	 loose

more rope ▼ — steer continuously → set loose, check outcome

El salto que importa es el último. Con los dos primeros niveles diriges de forma continua; con un agente autónomo lo dejas correr solo y revisas el resultado. Eso es una habilidad distinta, y es de lo que trata el resto de este capítulo.

Cuándo dejarlo correr vs. dirigir manualmente

La decisión de dejar que un agente corra sin supervisión se reduce a tres preguntas — una lista de verificación que revisas *antes* de alejarte.

- **Radio de impacto.** Si esto sale mal, ¿cuánto se rompe? Un pase de formateo por todo el código tiene un radio de impacto pequeño. Una migración de base de datos o un cambio a tu lógica de autenticación tiene uno grande. Cuanto mayor el radio, más debes dirigir a mano.
- **Reversibilidad.** ¿Puedes deshacerlo de forma barata? El trabajo en una rama git nueva con un árbol de trabajo limpio es reversible — `git restore .` y desaparece. El trabajo que borra datos, envía correos, cobra tarjetas o hace push a producción no lo es. Nunca dejes que un agente corra sin supervisión sobre algo que no puedas revertir.
- **Verificabilidad.** ¿Puedes saber si el resultado es correcto? Si “terminado” significa una suite de pruebas en verde o una página que visiblemente funciona, puedes verificarlo en segundos y un agente encaja perfecto. Si “terminado” depende de un juicio sutil — matemática de dinero, reglas de seguridad, condiciones de carrera — no puedes usar una verificación rápida como filtro, así que te quedas cerca.

Una forma simple de combinarlas: **déjalo correr cuando el trabajo es reversible y el resultado es barato de verificar; dirige a mano cuando no lo es.** Retoma la regla del capítulo de construir funcionalidades — cuanto mayor el radio de impacto, más pequeño el paso — y añade su hermana: cuanto mayor el radio de impacto, más corta la correa.

Delimitar una tarea para que no pueda desviarse

Un agente autónomo falla de una manera específica: se desvía. Pides una cosa, “útilmente” refactoriza otras tres, toca archivos que nunca mencionaste, y vuelve con un diff que no puedes leer. La solución no es un agente más inteligente — es un encargo más ajustado.

Una buena tarea autónoma tiene cuatro partes: un **objetivo** claro, **restricciones** explícitas, un **criterio de finalización** concreto, y **barandillas** que el agente puede verificar por sí mismo. Las barandillas son la parte más importante, porque son la cerca que el agente no puede saltar sin que tú lo notes.

Goal: Make the failing tests in tests/checkout/ pass.

Constraints:

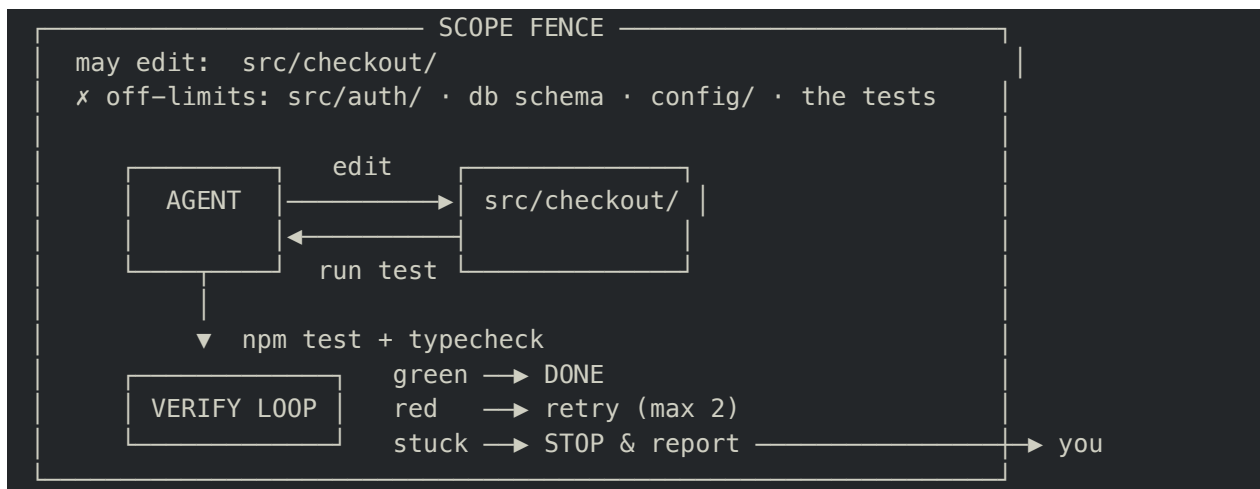
- Only edit files under src/checkout/. Do not touch src/auth/, the database schema, or anything in config/.
- Do not add new dependencies.
- Do not change the tests themselves – the tests define correct behavior.

Done when:

- `npm test tests/checkout/` is fully green.
- `npm run typecheck` passes.
- `git diff --stat` shows changes only under src/checkout/.

If you get stuck after two real attempts, stop and report what you tried and what's still failing. Don't paper over a failure to make the suite go green.

El encargo funciona porque encierra al agente: las restricciones delimitan lo que puede tocar, el criterio de finalización es un comando que puede ejecutar, y el bucle de verificación lo mantiene honesto:



Tres cosas hacen que este encargo sea difícil de desviar. Las restricciones dibujan un muro alrededor de los archivos que el agente puede tocar. El criterio de finalización es un comando que cualquiera puede ejecutar, no una sensación. Y el último párrafo le da permiso para *detenerse y preguntar* en vez de flagelarse — lo que evita el desastre en cámara lenta donde un agente sigue “arreglando” cosas durante veinte minutos y deja el árbol peor de como lo encontró.

Las pruebas son la cerca real. Un agente que puede ejecutar sus propias pruebas tiene un bucle de retroalimentación ajustado y honesto, y en su mayor parte se mantiene dentro de él; uno sin forma de verificar su trabajo te entregará con confianza algo roto. Si te llevas un solo hábito de este capítulo: **dale al agente una forma de verificarse a sí mismo, y haz que pasarla sea la definición de terminado.**

Agentes en segundo plano y en paralelo

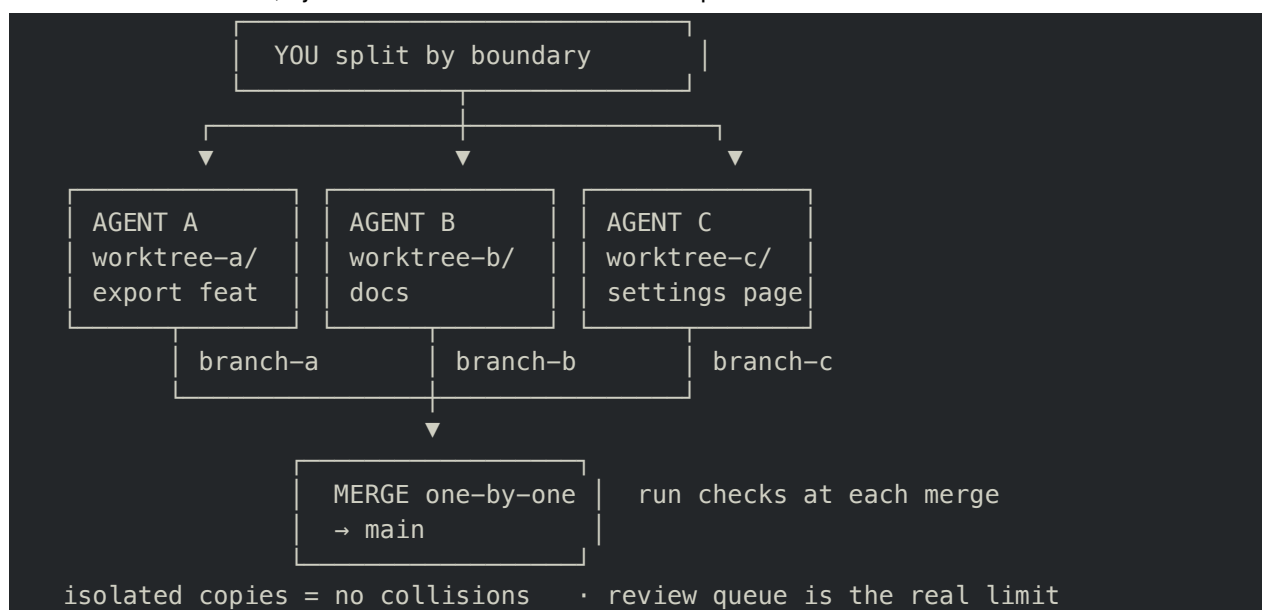
Una vez que confías en un agente para una tarea delimitada, el siguiente paso es obvio: ejecutar varios a la vez. Las herramientas modernas te permiten lanzar agentes en segundo plano — cada uno en su propia rama o worktree (una copia de trabajo separada del proyecto, para que las ediciones de un agente nunca choquen con las de otro) — de modo que tres o cuatro tareas delimitadas avanzan mientras haces otra cosa.

Esto es una aceleración real, pero solo para trabajo *independiente*. El costo contra el que estás negociando

es la coordinación. Dos agentes editando el mismo archivo producirán diffs en conflicto, y fusionarlos a mano puede costar más de lo que ahorraste.

- **Divide por límites, no por número de líneas.** Dale a cada agente un trozo que no se superponga con los demás — módulos separados, funcionalidades separadas, archivos separados. “El agente A hace la funcionalidad de exportación, el agente B escribe la documentación” es limpio. “Ambos agentes editan `app.js`” es un dolor de cabeza de fusión esperando a suceder.
- **Aísla los espacios de trabajo.** Ejecuta cada agente en su propia rama o worktree de git para que no puedan pisarse los cambios sin confirmar del otro. Fusiónalos de vuelta uno a la vez, ejecutando tus verificaciones en cada fusión.
- **Mantén el conteo honesto.** Sigues teniendo que revisar cada resultado. Cuatro agentes ejecutándose son cuatro diffs por leer y cuatro conjuntos de pruebas en los que confiar. Pasado un puñado, la cola de revisión — no los agentes — se convierte en tu cuello de botella.

Cada agente trabaja en su propio worktree aislado para que sus ediciones nunca choquen; los fusionas de vuelta uno a la vez, ejecutando verificaciones en cada puerta:



El error es tratar el paralelismo como gratuito. No lo es — has movido el trabajo de *escribir* a *revisar e integrar*. A menudo un gran intercambio, pero solo si de verdad haces la revisión.

Multiagente: expansión, adversario, síntesis

Hay un segundo tipo de “muchos agentes” que se ejecuta *dentro* de una sola tarea en lugar de a través de tareas separadas. En vez de que un agente haga todo, divides los roles:

- **Expansión (fan-out) para amplitud.** Varios agentes abordan el mismo problema desde ángulos diferentes — o exploran distintas partes de un código base grande — y luego informan de vuelta. Útil cuando quieres cobertura: “encuentra cada lugar donde llamamos a esta función obsoleta” se paraleliza bien.
- **Un revisor adversarial.** Un agente escribe; el único trabajo de un segundo agente es *atacar* la salida del primero — encontrar el bug, el caso límite faltante, el error no manejado. Un revisor apuntado a un diff con la instrucción “asume que esto está mal y averigua por qué” detecta cosas que el agente autor se autoconvenció de que estaban bien.
- **Síntesis.** Un pase final recopila los hallazgos y produce un resultado coherente — el código funcional más una nota breve sobre lo que el revisor señaló y cómo se resolvió.

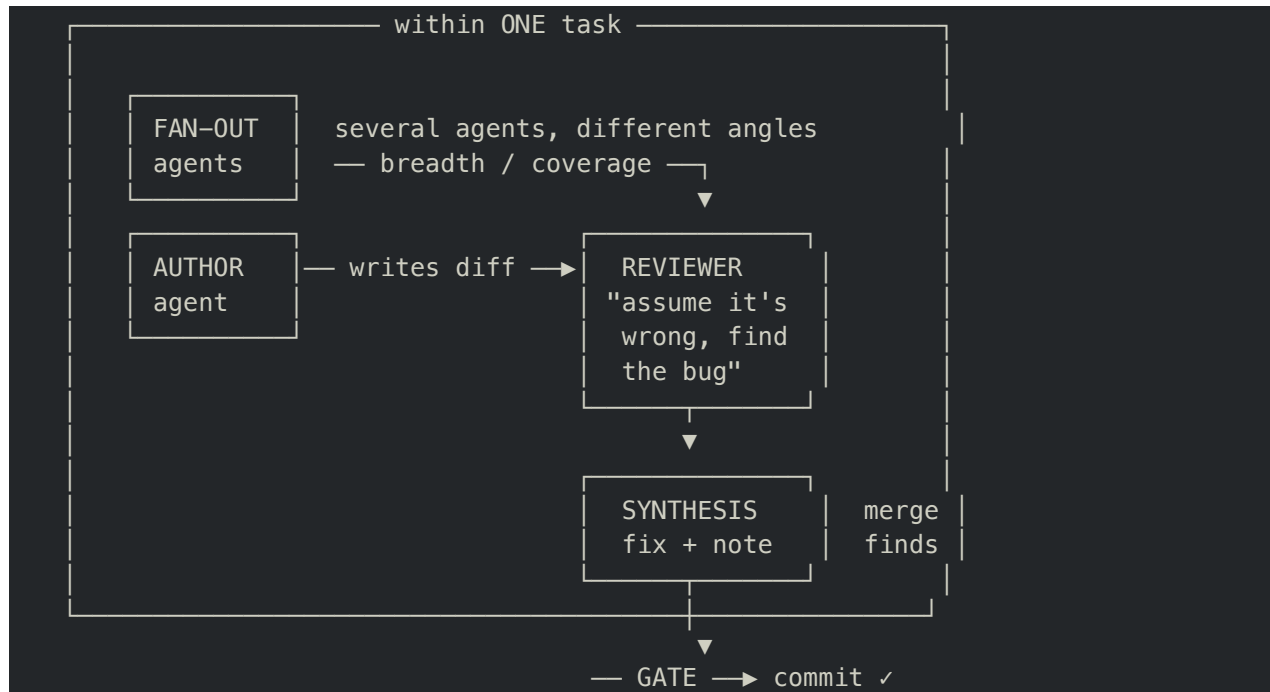
Reviewer agent brief:

You are reviewing a diff written by another agent. Assume it contains at least one bug and find it. Specifically check:

- Does it actually satisfy the original goal, or just pass the tests?
- Edge cases: empty input, nulls, the largest realistic input.
- Anything deleted or changed that the task didn't ask for.

Report concrete problems with file and line. Do not rewrite the code — just report. If you find nothing after a genuine look, say so plainly.

Los tres roles forman una tubería con una puerta al final: nada llega a un commit hasta que el revisor adversarial ha hecho su pase y la síntesis ha resuelto lo que encontró.



El revisor adversarial vale la pena configurarlo incluso cuando eres una sola persona sin una flota de agentes — porque el autor de un cambio es la peor persona para encontrar sus fallos, humano o IA. Un segundo pase con un prompt hostil es un seguro barato.

Revisar salida a gran escala

Aquí está el problema honesto: cuando los agentes están escribiendo cientos de líneas a través de varias tareas, no puedes leer cada línea de la forma en que revisarías un diff de diez líneas. Fingir que puedes lleva o bien al sellado automático (apruebas sin mirar) o a la parálisis (nunca confías en nada). Ninguno de los dos logra enviar algo.

El camino a seguir es filtrar en unas pocas capas en lugar de leerlo todo:

- **Las pruebas son el piso mínimo.** Ninguna salida de agente se fusiona sin que las pruebas relevantes estén en verde y el verificador de tipos esté limpio. Esto no es negociable y es automático — la máquina hace esta parte.
- **Revisa puntualmente las partes riesgosas.** No puedes leerlo todo, así que lee las partes donde el radio de impacto es alto: cualquier cosa que toque autenticación, dinero, borrado de datos o llamadas externas. Hojea el resto buscando las señales de alerta del capítulo de construcción de funcionalidades — archivos que no debería haber tocado, cosas borradas silenciosamente, una petición diminuta que produjo un diff enorme.

- **Lee el diff, no el código base.** `git diff --stat` te dice *qué* cambió y cuánto en segundos. Una tarea delimitada a un solo módulo que muestra cambios en seis es una señal para mirar más de cerca, antes de leer una sola línea.
- **Ejecútalo.** Una suite de pruebas en verde no es una aplicación funcionando. Haz clic en la cosa. La verificación final es siempre la misma de todos los demás capítulos: ¿realmente hace lo que pediste, delante de ti?

Delimitar bien la tarea rinde frutos dos veces aquí. Una tarea que delimitaste con precisión produce un diff que realmente puedes revisar; una tarea que dejaste suelta produce uno que no puedes. El problema de la revisión se resuelve en su mayor parte río arriba, en el encargo.

Costo y bucles descontrolados

Un agente autónomo funciona iterando: intenta, verifica, vuelve a intentar. Ese bucle es todo el punto — y también es lo que puede silenciosamente disparar una factura o girar para siempre.

Esto conecta con la idea de seguridad de costos de antes: **una herramienta de suscripción de tarifa plana falla deteniéndose; un agente de API pagada por medición falla cobrando.** Cuando un asistente de código con suscripción alcanza su límite, simplemente no hace más hasta que la ventana se restablece. Cuando un script que escribiste contra una API pagada queda atrapado en un bucle de reintentos, sigue llamando — y cada llamada cuesta dinero — hasta que lo notas o tu tarjeta llega al máximo.

Así que las reglas son directas:

- **Nunca ejecutes un bucle de agente sin límite contra una API pagada.** Si has escrito tu propio bucle que llama a una API de LLM (Large Language Model, modelo de lenguaje extenso — el tipo de IA detrás de estas herramientas de código), debe tener un tope estricto — un número máximo de iteraciones, un límite de gasto, un tiempo de espera. Sin excepciones, sin “lo voy a vigilar.” No lo harás.
- **Dale a cada bucle una salida.** “Detente después de N intentos,” “detente si las pruebas siguen fallando dos veces,” “detente si llevas diez minutos ejecutándose.” Un agente sin condición de parada es un bug, no una funcionalidad.
- **Vigila las primeras ejecuciones.** Antes de confiar en que un agente se ejecute en segundo plano, observa unas pocas ejecuciones en primer plano de principio a fin. Estás verificando que *converge* — que se acerca a terminar, no que queda atascado dando vueltas alrededor del mismo arreglo.
- **Prefiere la herramienta con barandillas.** Un asistente gestionado con límites incorporados es más seguro de dejar corriendo que un bucle de API en bruto que escribiste a medianoche. Que la suscripción falle cerrando es una funcionalidad; que la API falle abriendo es un peligro.

Límites honestos

Los agentes son la forma más poderosa de apalancamiento en este libro, que es exactamente por qué vale la pena ser claro sobre lo que son y lo que no son.

No son autonomía que puedas dejar de entender. En el momento en que ya no puedes responder “¿qué se supone que hace este código, y cómo sabría si está mal?”, has cruzado de usar apalancamiento a esperar tener suerte. Cada capa de este capítulo — delimitación, barandillas, pruebas, revisión puntual, topes de costo — existe para mantenerte del lado de la comprensión de esa línea mientras dejas que el agente haga mucho más de la escritura.

El techo de habilidad es el mismo que el resto del libro: tu capacidad de *decir con precisión lo que quieres y reconocer cuándo lo lograste*. Un agente multiplica esa capacidad; no la reemplaza. Las personas que se queman no son las que usaron agentes — son las que usaron agentes para dejar de pensar. No seas esa persona. Delega el trabajo; conserva el juicio.

Resumen y práctica

Puntos clave

- Los agentes multiplican tu capacidad de decir con precisión lo que quieres y de reconocer cuándo lo lograste — no la reemplazan.
- Delimitar bien rinde frutos dos veces: una tarea bien delimitada produce un diff que realmente puedes revisar, mientras que una suelta produce uno que no puedes.
- Revisa el diff, no el código base — `git diff --stat` señala la expansión de alcance en segundos, y una suite de pruebas en verde nunca sustituye ejecutar la cosa.
- Nunca ejecutes un bucle de agente sin límite contra una API pagada; dale a cada bucle un tope estricto — iteraciones máximas, un límite de gasto, o un tiempo de espera.
- Observa las primeras ejecuciones en primer plano para confirmar que el agente *converge* antes de confiar en que se ejecute en segundo plano.

Pruébalo

Toma una tarea pequeña y bien delimitada y entrégala a un agente con una condición de parada explícita (“detente después de 3 intentos” o “detente si las pruebas siguen fallando dos veces”). Observa la ejecución de principio a fin en primer plano, luego revisa solo el diff con `git diff --stat` seguido de `git diff`. Si los cambios tocan más archivos de los que delimitaste, esa es tu señal para ajustar el encargo y ejecutarlo de nuevo.

Prompt del capítulo

```
I'm giving you an autonomous task. Obey these guardrails exactly:
- SCOPE: only touch [list the files/modules]. If the fix needs more,
  STOP and tell me instead of expanding scope.
- GOAL: [one concrete, checkable outcome].
- STOP CONDITIONS: stop after 3 attempts, OR if tests fail twice in a
  row, OR if you've made changes outside the scope above.
- COST: do not call any paid API in a loop without a hard cap.
- REPORT: when you stop, show me `git diff --stat`, the test result,
  and one sentence on whether you converged or got stuck.
Begin, and narrate each attempt so I can watch you converge.
```