

Vibe Handbook

Vibe Handbook

Sumário

Git & GitHub	5
O que é controle de versão (e por que você precisa dele)	6
Repositórios, commits e histórico	6
Branches: experimentando sem medo	7
Vendo o que realmente mudou (diff)	8
O botão do “ih, não”: undo, revert, reset	8
Remotos e GitHub: seu projeto na nuvem	8
Pull requests e issues	9
.gitignore e nunca fazer commit de segredos	9
Seu primeiro repo, do início ao fim	10
Deixando a IA dirigir (enquanto você continua no comando)	10
Recapitulação e prática	10
A Linha de Comando	12
O que o terminal e o shell realmente são	12
Lendo o prompt	13
Os comandos do dia a dia	13
Rodando um programa e um servidor de desenvolvimento	14
Lendo a saída e os códigos de saída	14
Interrompendo um processo com Ctrl-C	15
PATH e “command not found”	15
Rodando comandos que a IA te passa — com segurança	15
Gerenciadores de pacotes que você vai encontrar	16
Recapitulação e Prática	16
Como a Web Funciona	17
Os dois personagens: cliente e servidor	17
O que uma URL realmente é	18
O ciclo de request e response	18
Métodos HTTP: os verbos	19
Códigos de status: deu certo?	19
Headers e cookies	20
HTTPS e o cadeado	20
De um nome a um servidor: DNS	20
Como isso se encaixa no seu app	21
Recapitulando e Praticando	21
APIs, JSON e Dados	22
O que uma API realmente é	23
Endpoints e REST, em termos simples	23
O que é JSON	24
Os tipos de dados que você vai encontrar	24

Uma requisição e resposta reais	25
Chaves de API e cabeçalhos de autenticação	25
Limites de taxa e a armadilha da conta chocante	26
Como ler a documentação de uma API	26
Deixando a IA fazer as chamadas — enquanto você confere o formato	26
Recapitulando e Praticando	27
Anatomia de um App	28
O panorama geral	28
Frontend, backend, banco de dados, hospedagem	29
HTML, CSS e JavaScript	29
Pacotes e dependências	30
Variáveis de ambiente e configuração	31
A forma de um projeto	31
Como uma requisição flui	32
Recapitulação e Prática	33
A mentalidade do vibe coding	34
O que é vibe coding, de fato	34
A mudança de mentalidade	35
O ciclo: intenção → gerar → revisar → refinar	36
Onde brilha	37
Onde falha	37
Armadilhas comuns	38
Bom senso é a nova habilidade central	38
Expectativas realistas	39
Recap and Practice	39
Configurando Seu Ambiente de Trabalho com IA	40
Escolha um Editor ou Assistente Nativo de IA	41
Estruture Seu Projeto Para Que a IA Consiga Navegar Nele	41
Escreva um Arquivo de Contexto (AGENTS.md)	42
Faça do Controle de Versão um Reflexo	43
Mantenha o Ciclo de Feedback Curto	43
Gerencie Segredos com Cuidado	44
Uma Configuração Local Sensata Torna as Mudanças da IA Seguras	45
Diagnósticos ao Vivo (LSP)	45
Estendendo Seu Agente	46
macOS vs. Windows	46
TUI vs. GUI	47
Engenharia de Harness	47
Segurança de Custos: CLI vs. API	48
Recapitulação e Prática	49
Prompt Como um Engenheiro	50
Dê Contexto Antes de Dar a Tarefa	50
Especifique Entradas, Saídas e Casos de Borda	51
Trabalhe em Passos Pequenos	52
Peça um Plano Antes do Código	52
Itere em Diffs, Não em Reescritas	53
Forneça Exemplos	53
Diga à IA o Que Não Fazer	54
Gerencie o Contexto Como um Recurso	54
Antipadrões Comuns a Evitar	55
A Mentalidade Central	55

Recapitulação e Prática	56
Da Ideia à Especificação em Minutos	57
Por Que uma Spec, Não Só Vibração	57
Passo 1: Esclareça o Problema	58
Passo 2: Defina o Escopo e os Não-Objetivos	59
Passo 3: Escreva Histórias de Usuário	59
Passo 4: Rascunhe um PRD Leve	60
Passo 5: Esboce o Modelo de Dados	61
Passo 6: Divida a Spec em Tarefas do Tamanho de uma Vibe	61
Mantenha a Spec Viva	62
Uma Armadilha Comum: Especificar Soluções, Não Problemas	62
A Recompensa	63
Recapitulação e Prática	63
Escolhendo Sua Stack: Linguagens em Perspectiva	64
JavaScript / TypeScript	65
Python	65
Go	66
Rust	66
Uma Palavra Rápida Sobre as Outras	67
Comparação em Perspectiva	68
O Custo Real de uma Escolha Exótica	68
Escolha por Objetivo	69
Recapitulação e Prática	70
Construindo Funcionalidades, Vibe por Vibe	71
O Loop de Construção	71
O Que Significa “Fatia Vertical”	72
Um Exemplo Trabalhado: Uma Funcionalidade de “Favoritos”	73
Lendo um Diff Sem Entender Cada Linha	75
Quando Deixar a IA Correr Livre vs. Guiar Manualmente	75
Mantenha o App Sempre Funcionando	75
Fazendo Commits em Pequenos Incrementos	76
Quando Uma Fatia É Grande Demais	76
O Ponto Principal	76
Recapitulação e Prática	76
Depurando Com IA	78
A mentalidade central	78
Uma checklist de depuração	79
Alimentando a IA com o contexto certo	80
Lendo erros que você não entende	81
Hipóteses, registros e busca binária	81
Causa raiz versus sintoma	82
Quando a IA gira em círculos	83
Entendendo a correção antes de mantê-la	83
O ponto principal	83
Recapitulação e Prática	84
Infraestrutura de servidor	85
Os cinco lugares onde o código pode viver	85
Alguns termos técnicos, decodificados	86
A tabela de trade-offs	87
Quando escolher qual	87

Como a IA monta deploys e configuração	88
Um caminho padrão recomendado para o construtor solo	89
O que “gerenciado” realmente significa (e o que ainda é seu)	90
Uma checklist rápida de pré-lançamento	90
O resumo	91
Recapitulação e Prática	91
Infraestrutura de Banco de Dados	92
As Três Formas de Armazenamento de Dados	93
Gerenciado vs Auto-hospedado	94
Fundamentos de Design de Schema	94
Migrações: Mudando o Schema com Segurança	95
Indexação	96
Cache	97
Filas	97
Erros Comuns de Iniciante	97
Direcionando a IA para Construir Sua Camada de Dados	98
Recapitulação e Prática	98
Guardrails, Testes e Revisões	99
Por que você precisa de guardrails ao dirigir a IA	100
Testes: sua rede de segurança	100
Verificação de tipos e linting: erros pegos de graça	101
Limitando o escopo das mudanças da IA	102
Lendo diffs com espírito crítico	102
Portões de revisão: uma checklist antes de mesclar	103
Fundamentos de CI: torne as verificações automáticas	103
Hábitos seguros por padrão	104
Quando um guardrail grita: faça a triagem antes de corrigir	104
Recapitulação e prática	105
Colocando em Produção	106
Escolha um Host e Faça o Deploy	107
Domínios Personalizados e DNS	108
Variáveis de Ambiente e Segredos	109
Configurando Pagamentos	109
Staging e Rollbacks	110
Monitoramento e Rastreamento de Erros	111
Performance e Cache	111
Checklist Pré-Lançamento	112
Depois do Lançamento	112
Recapitulação e Prática	112
Estudos de Caso: Do Zero ao Lançamento	114
Estudo de Caso 1: Uma Página de Vendas Multilíngue com Pagamentos	114
Estudo de Caso 2: Uma Pequena Ferramenta SaaS com Autenticação e Banco de Dados	116
Estudo de Caso 3: Um Utilitário Pessoal de Automação	118
Estudo de Caso 4: Um Site de Conteúdo que Você Não Precisa Cuidar	120
O que os quatro têm em comum	122
Recapitulação e Prática	122
Segurança para vibe coders	123
Por que código gerado por IA é inseguro por padrão	124
Injeção: quando a entrada vira código	124
Segredos e chaves de API: a armadilha da chave exposta	125

Autenticação vs autorização: o endpoint que qualquer um pode chamar	126
Manipulação de arquivos e uploads	127
Risco de dependências: o pacote que a IA inventou	127
O portão de revisão de segurança	128
Complacência da automação, edição segurança	129
Recapitulação e prática	129
Orquestração de Agentes	130
As Três Camadas	131
Quando Deixar Rodar vs. Direcionar Manualmente	131
Delimitando uma Tarefa Para Que Ela Não Se Perca	132
Agentes em Segundo Plano e em Paralelo	133
Multi-Agente: Fan-Out, Adversário, Síntese	133
Revisando Saídas em Escala	134
Custo e Loops Descontrolados	135
Limites Honestos	135
Recapitulação e Prática	136

Git & GitHub



Se você só vai aprender uma ferramenta antes de começar a fazer vibe coding, que seja esta. Git é o que permite que você seja corajoso. É por causa dele que você pode dizer a uma IA “reescreva este arquivo inteiro” e não perder o sono, porque você sempre pode voltar para a versão que funcionava.

Aqui vai a versão honesta de por que isso importa. Quando você direciona uma IA para construir software, ela *vai* quebrar coisas — regularmente. Ela vai reescrever um arquivo com toda a confiança do mundo, apagar a parte que estava funcionando e te entregar uma bagunça. A diferença entre uma tarde frustrante e uma semana perdida é ter, ou não, uma rede de segurança. Git é essa rede. Este capítulo ensina do zero, sem exigir experiência prévia com programação.

O que é controle de versão (e por que você precisa dele)

Imagine um videogame com pontos de salvamento. Você joga uma parte complicada, salva, tenta algo arriscado. Se der errado, você recarrega o último save em vez de recomeçar o jogo inteiro. **Controle de versão são pontos de salvamento para o seu projeto.**

Sem ele, seu projeto tem exatamente um estado: o que ele é agora. Se a IA bagunçar seu código, a versão boa se foi. Com controle de versão, você mantém uma linha do tempo completa de todo estado salvo, e pode voltar para qualquer um deles.

Git é a ferramenta de controle de versão mais usada. Ela roda no seu computador e registra silenciosamente instantâneos (“snapshots”) do seu projeto sempre que você pedir. Algumas coisas que ela te dá de graça:

- **Uma rede de segurança** — desfazer uma mudança ruim mesmo horas depois.
- **Um histórico** — ver exatamente o que mudou, quando e por quê.
- **Confiança** — experimentar livremente, porque nada nunca se perde de verdade.

Esse último ponto é o verdadeiro prêmio. Vibe coding funciona melhor quando você se move rápido e testa coisas. Git é o que torna mover-se rápido seguro, em vez de imprudente.

Imagine o histórico do seu projeto como uma fileira de pontos de salvamento aos quais você sempre pode voltar:

```
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
```

Repositórios, commits e histórico

Três palavras que você vai ouvir o tempo todo. Vamos desmistificá-las.

Um **repositório** (ou “repo”) é apenas uma pasta que o Git está observando. Você transforma uma pasta comum em um repo uma vez, e a partir daí o Git rastreia tudo dentro dela.

Um **commit** é um ponto de salvamento. É um instantâneo de todos os arquivos em um determinado momento, mais uma mensagem curta sua descrevendo o que mudou (“adicionei a página de login”, “corrigi o botão de cadastro quebrado”). Você faz commits o tempo todo — pense em cada um como um checkpoint para o qual você pode querer voltar.

O **histórico** é a lista ordenada de todos os seus commits. É a linha do tempo. Você pode percorrê-la, ler o que cada mudança foi, e viajar de volta para qualquer ponto.

Veja como isso funciona na prática. Não se preocupe em decorar — na maior parte do tempo você vai deixar a IA rodar esses comandos para você — mas acompanhe a leitura para que as palavras deixem de ser um mistério:

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

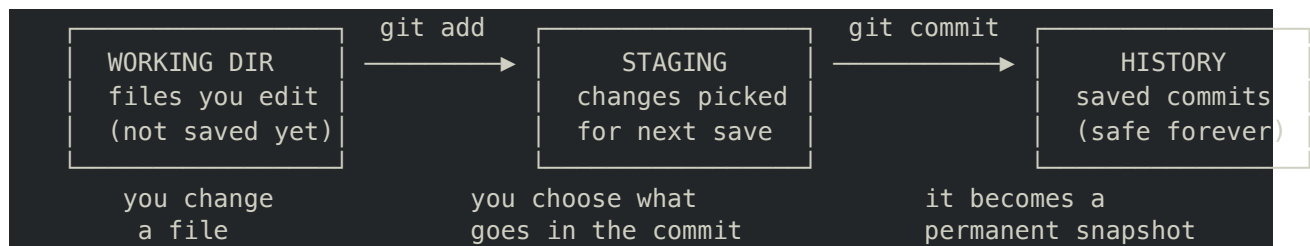
```
# See which files have changed since your last save point
git status

# 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 .` é você dizendo “inclua todas as minhas mudanças no próximo ponto de salvamento.” `git commit` é o salvamento em si. O `-m "...` é o seu bilhete para o seu eu do futuro. Escreva notas que você entenderia um mês depois.

Esses dois comandos movem seu trabalho por três estágios. Um arquivo que você edita não está salvo até ter percorrido todo o caminho até a direita:



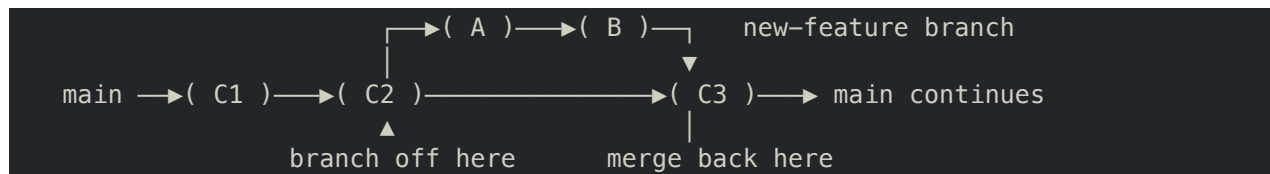
Branches: experimentando sem medo

Uma **branch** (ramo) é uma cópia paralela do seu projeto onde você pode experimentar. Seu trabalho principal fica em uma branch geralmente chamada `main`. Quando você quer tentar algo — um recurso novo, uma reescrita arriscada feita por IA — você cria uma nova branch, faz o trabalho lá, e sua `main` permanece intocada e segura.

Se o experimento funcionar, você faz o merge (mescla) de volta para a `main`. Se não funcionar, você descarta a branch e a `main` nunca soube que aquilo aconteceu.

É exatamente assim que você deve tratar mudanças grandes feitas por IA. “Construa todo o fluxo de pagamentos” merece sua própria branch. Se a IA produzir algo bom, você faz o merge. Se produzir um desastre, você apaga a branch e não perdeu nada.

Visualmente, uma branch se separa da `main`, cresce por conta própria e depois se funde de volta:



```
# 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
```

Vendo o que realmente mudou (diff)

Antes de salvar uma mudança — e *especialmente* antes de aceitar uma mudança feita por uma IA — você quer ver exatamente o que é diferente. Isso é um **diff**. Ele te mostra, linha por linha, o que foi adicionado (geralmente em verde) e o que foi removido (geralmente em vermelho).

Esse é um dos seus hábitos mais importantes. Toda a filosofia do vibe coding é *entender o que você entrega*. Você não precisa digitar o código, mas precisa olhar o que mudou e entender minimamente. O diff é onde você faz isso.

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

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

Quando a IA diz “pronto, atualizei o arquivo”, seu próximo passo é olhar o diff. Se você ver que ela apagou algo que você precisava, você percebe *agora* — não depois que já foi para produção.

O botão do “ih, não”: undo, revert, reset

Essa é a parte que transforma o Git de uma tarefa chata em um superpoder. Quando algo dá errado — e vai dar — veja como reverter.

- **Descartar mudanças não salvas** — você editou (ou a IA editou) alguns arquivos e piorou as coisas, mas ainda não fez o commit. Você pode descartar tudo e voltar ao seu último commit bom.
- **Reverter um commit** — você salvou uma mudança e depois percebeu que era ruim. `git revert` cria um *novo* commit que desfaz o commit ruim, mantendo seu histórico honesto.
- **Voltar para um ponto anterior** — você quer rebobinar seu projeto para como ele estava vários commits atrás.

```
# 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>
```

Um aviso: `git reset --hard` descarta trabalho permanentemente. É a ferramenta certa às vezes, mas é aquela com a qual você precisa ter cuidado. Na dúvida, pergunte à IA “o que exatamente esse comando vai fazer com meus arquivos?” antes de rodá-lo.

Remotos e GitHub: seu projeto na nuvem

Tudo até agora acontece no seu próprio computador. **GitHub** é um site que armazena uma cópia do seu repo na nuvem. Essa cópia é chamada de **remoto** (remote).

Por que se importar? Três razões: é um backup caso seu laptop morra, é como você compartilha código com outras pessoas (ou com ferramentas de IA e serviços de deploy), e é onde a colaboração acontece.

Você conecta seu repo local a um remoto uma vez, e depois faz **push** (envia seus commits para o GitHub) e **pull** (traz commits de volta do GitHub).

Pense em duas cópias do seu projeto — uma no seu laptop, outra na nuvem — mantidas sincronizadas por push e 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
```

Pense em push como “salvar na nuvem” e pull como “pegar o que há de mais recente na nuvem.”

Pull requests e issues

Dois recursos do GitHub que você vai conhecer rapidamente.

Um **pull request** (PR) é uma proposta de fazer o merge de uma branch em outra, com um espaço embutido para revisar as mudanças antes que elas entrem. Mesmo trabalhando sozinho, PRs são úteis: elas te dão uma tela única mostrando cada mudança, para você revisar o trabalho da IA antes de fazer o merge para a main. É o hábito do diff, formalizado.

Uma **issue** é uma anotação rastreada sobre algo a fazer ou corrigir — um bug, uma ideia de recurso, um TODO. Issues são como você mantém uma lista de tarefas anexada ao próprio projeto, em vez de espalhada na sua cabeça.

.gitignore e nunca fazer commit de segredos

Alguns arquivos *nunca* devem entrar no Git. Os principais:

- **Segredos** — chaves de API (Application Programming Interface, ou Interface de Programação de Aplicações), senhas, credenciais de banco de dados. Se você fizer commit de um segredo no GitHub, trate-o como vazado, mesmo que apague depois. O histórico lembra.
- **Lixo** — arquivos temporários, dependências baixadas (como `node_modules/`), saída de build. Enormes e inúteis de rastrear.

Você controla isso com um arquivo chamado `.gitignore`. Você lista padrões nele, e o Git finge que esses arquivos não existem. Isso é fundamental para quem faz vibe coding, porque ferramentas de IA adoram criar arquivos de configuração com chaves reais dentro, e é fácil fazer commit de um por acidente.

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

A linha `.env` é a importante — é onde os segredos costumam morar. Adicione o `.gitignore` *antes* do seu primeiro commit, e crie o hábito de perguntar à IA: “tem alguma coisa neste commit que não deveria ser pública?”

Seu primeiro repo, do início ao fim

Aqui está o ciclo completo em um só lugar. Crie uma pasta, transforme-a em um repo, salve seu trabalho e envie para o 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
```

Pronto. Agora você tem um projeto com uma rede de segurança e um backup na nuvem. Toda vez que fizer progresso, repita o ritmo `add` → `commit` → `push`.

Deixando a IA dirigir (enquanto você continua no comando)

Aqui vai a parte libertadora: você não precisa decorar nenhum desses comandos. Você pode dizer à IA “faça o commit disso com uma mensagem sensata” ou “crie uma branch para o novo recurso e mude para ela”, e ela vai rodar os comandos certos. Essa é uma forma perfeitamente boa de trabalhar.

A pegadinha — e é a mesma pegadinha do resto deste livro — é que você continua responsável por entender *o que* ela fez, mesmo que não tenha digitado. O caminho do meio que realmente funciona:

- **Deixe a IA rodar os comandos**, mas peça para ela te explicar o que cada um faz em português simples nas primeiras vezes.
- **Leia o diff você mesmo** antes de qualquer commit. Isso não é negociável. É assim que você pega a IA apagando algo que não deveria.
- **Faça commits com frequência, em pedaços pequenos.** Muitos pontos de salvamento pequenos são melhores que um gigante, porque rebobinar fica preciso em vez de tudo-ou-nada.
- **Crie uma branch antes de qualquer coisa grande.** Diga à IA para criar uma branch antes de uma reescrita importante, para que a `main` permaneça segura.
- **Quando um comando parecer destrutivo** (qualquer coisa com `reset --hard`, `force-push`, ou apagar branches), pergunte “o que exatamente isso faz?” antes de aprová-lo.

Você é o tech lead. A IA é sua colega de equipe rápida, capaz e ocasionalmente imprudente. Git é o processo que impede essa colega de causar dano real. Aprenda o suficiente para dirigi-la com confiança, e você nunca mais vai temer o botão “reescreva tudo” — porque você sempre terá um caminho de volta.

Recapitulação e prática

Principais pontos

- Git são **pontos de salvamento para o seu projeto** — uma linha do tempo completa para a qual você pode voltar, e é isso que permite soltar a IA sem medo.
- O ritmo central é `add` → `commit` → `push`: prepare suas mudanças (stage), salve-as como um commit com uma mensagem clara, e envie-as para o GitHub para backup e compartilhamento.
- **Crie uma branch antes de qualquer coisa grande, e leia o diff antes de cada commit** — é assim que você pega a IA apagando algo que não deveria.
- Um `.gitignore` mantém segredos (como `.env`) e lixo (como `node_modules/`) fora do seu histórico. Um segredo commitado no GitHub fica vazado para sempre, mesmo que você apague depois.
- Você pode deixar a IA rodar os comandos, mas continua responsável por entender o que cada um fez — especialmente os destrutivos, como `reset --hard`.

Pratique

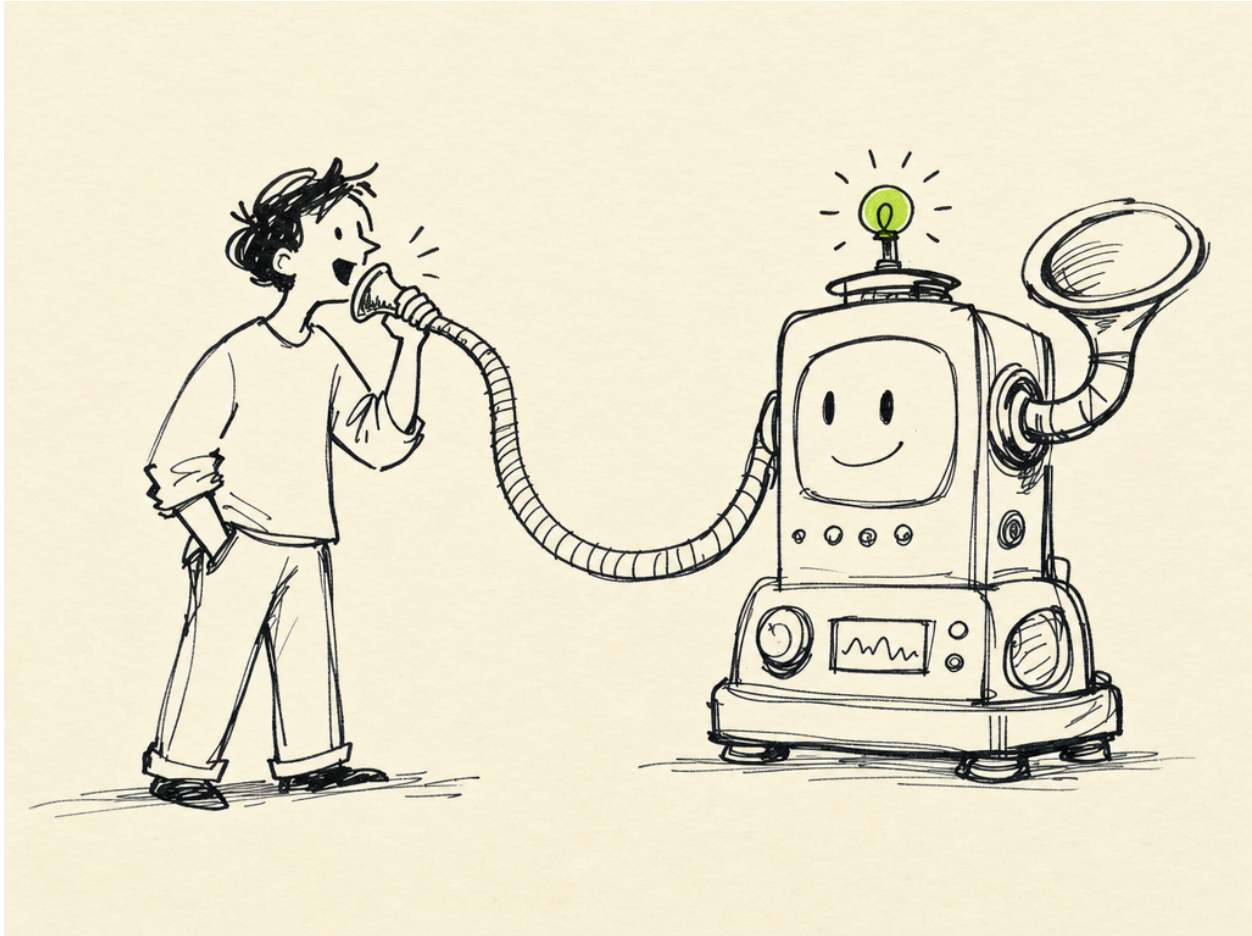
Crie uma pasta totalmente nova e rode o ciclo completo você mesmo, na mão: `git init`, crie um `.gitignore` com `.env` dentro, adicione um arquivo simples `notes.txt`, depois `git add .` e `git commit -m "Initial commit"`. Agora rode `git log --oneline` para ver seu ponto de salvamento, edite o arquivo, e rode `git diff` para ver o Git te mostrar exatamente o que mudou. Você acabou de usar todos os hábitos centrais deste capítulo em um projeto onde nada pode dar errado.

Prompt do 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).
```

A Linha de Comando



Na primeira vez que você vê uma ferramenta de IA de programação trabalhando, uma janela preta se enche de texto que passa rolando na tela, com um cursor piscando esperando você digitar. Essa janela é a linha de comando. Para muita gente que não programa, é a parte mais assustadora de toda a jornada — parece coisa de filme, como se um movimento errado fosse apagar o disco rígido inteiro. Não vai. A linha de comando é só um jeito diferente de dizer ao computador o que fazer: com palavras em vez de cliques.

Este capítulo desmistifica isso. Você não precisa decorar uma centena de comandos. Precisa entender o que está acontecendo, reconhecer um punhado de comandos do dia a dia e saber como rodar com segurança aqueles que uma IA te passa. Isso já basta para parar de ter medo e começar a ser eficaz.

O que o terminal e o shell realmente são

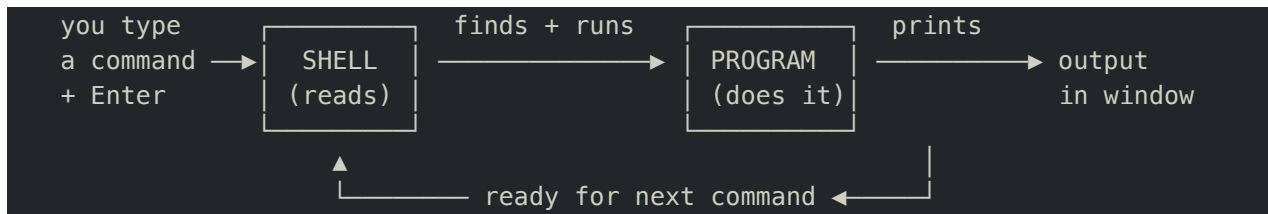
Quando você usa um aplicativo normal, clica em botões e o aplicativo traduz seus cliques em instruções para o computador. A linha de comando pula os botões. Você digita uma instrução, aperta Enter, e o computador a executa.

Duas palavras que você vai ouvir bastante:

- **Terminal** — a janela onde você digita. É só uma caixa de texto com histórico. (No Mac é o “Terminal” ou “iTerm”; no Windows é o “PowerShell” ou “Windows Terminal”; no Linux geralmente é só “Terminal”).
- **Shell** — o programa *dentro* dessa janela que lê o que você digita e executa. O mais comum se chama **bash**; os Macs usam um chamado **zsh**. Eles se comportam de forma quase idêntica para

tudo o que este livro cobre.

Aqui está o ciclo, toda vez que você aperta Enter: o shell lê o que você digitou, encontra e executa o programa correspondente, e imprime de volta na janela o que quer que esse programa tenha produzido.



Por que as ferramentas de IA de programação vivem aqui? Porque a linha de comando é o painel de controle universal do desenvolvimento. Instalar ferramentas, iniciar seu aplicativo, rodar testes, publicar na internet — tudo isso acontece por meio de comandos digitados. Um agente de IA funciona *digitando os mesmos comandos que você digitaria*, só que mais rápido e sem erros de digitação. Quando você o vê trabalhando, está vendo-o usar exatamente a mesma porta que você vai usar.

Lendo o prompt

Antes de você digitar qualquer coisa, o shell mostra um **prompt** — um pequeno trecho de texto terminado em um símbolo, esperando por entrada. Pode se parecer com isto:

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

Lendo da esquerda para a direita, isso diz: seu nome de usuário (mini), seu computador (laptop), onde você está atualmente (a pasta ~/projects/my-app), e depois um \$ significando “sua vez.” O ~ é um atalho para a sua pasta pessoal (home). O símbolo no final costuma ser \$ ou %; às vezes >. Não digite o símbolo — ele é só o shell avisando que está pronto.

Isso importa mais do que parece. A maior parte da confusão na linha de comando vem de não saber *onde* você está. O prompt te conta isso, toda vez.

Os comandos do dia a dia

Aqui estão os comandos que você vai ver o tempo todo. Leia-os como frases: um verbo, às vezes seguido de um alvo.

```
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
```

Algumas observações que evitam dores de cabeça de verdade:

Seus arquivos vivem em uma árvore de pastas aninhadas, e o cd é como você caminha por ela. Aqui vai um exemplo pequeno, com o pwd mostrando onde você estaria parado se tivesse feito cd para dentro de 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/

```

- **O cd é como você se move por aí.** Combinado com pwd e ls, é 80% de tudo que você vai fazer. Você entra na pasta certa, olha ao redor, e então roda alguma coisa.
- **Caminhos podem ser relativos ou absolutos.** cd projects é relativo a onde você está; cd /Users/mini/projects é o endereço completo a partir do topo.
- **O rm é permanente.** Não existe Lixeira nem Lixo na linha de comando. Quando você usa rm num arquivo, ele se vai — sem “tem certeza?”, sem desfazer. Tenha cuidado especial com rm -rf, que apaga uma pasta e tudo dentro dela sem perguntar. Se você não tiver certeza do que um comando rm vai apagar, não o rode. Peça para a IA explicar primeiro.

Rodando um programa e um servidor de desenvolvimento

Construir software significa rodá-lo. Na maior parte do tempo você vai digitar o nome de um programa seguido do que quer que ele faça.

```

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

```

Esse último é especial. Um **servidor de desenvolvimento** (dev server) é um programa que roda seu aplicativo no seu próprio computador para que você possa vê-lo em um navegador, em um endereço como http://localhost:3000. O ponto-chave a entender: ele **continua rodando**. O terminal vai parecer “travado” — ele para de te devolver um prompt novo e, em vez disso, mostra um fluxo contínuo de atividade. Isso não é travamento; é o programa funcionando. Seu aplicativo está vivo enquanto esse comando estiver rodando. Abra seu navegador, visite o endereço que ele imprime, e você verá seu aplicativo.

Quando terminar, você o interrompe (próxima seção).

Lendo a saída e os códigos de saída

Depois que um comando roda, ele geralmente imprime alguma coisa. Aprender a ler essa saída é uma habilidade de verdade, porque é assim que você e a IA descobrem se as coisas funcionaram.

- A **saída normal** é só informação — arquivos listados, mensagens de progresso, um servidor dizendo que iniciou.
- Os **erros** geralmente avisam claramente: Error:, command not found, permission denied, cannot find module. As palavras depois dos dois-pontos são sua pista. Copie o erro inteiro e cole para sua IA — é a coisa mais útil que você pode entregar quando algo quebra.

Nos bastidores, todo comando termina com um **código de saída** invisível: 0 significa sucesso, qualquer outra coisa significa um problema. Raramente você precisa checar isso à mão, mas vai ouvir menções a ele. Você pode ver o último assim:

```

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

```

Se um comando falhou silenciosamente e você não tem certeza, echo \$? te diz se ele realmente teve sucesso.

Interrompendo um processo com Ctrl-C

Aquele servidor de desenvolvimento continua rodando até você pará-lo. O “pare” universal é o **Ctrl-C** — segure a tecla Control e aperte C. Isso envia um educado sinal de “por favor pare agora” para o que quer que esteja rodando, e você recupera seu prompt.

- **Ctrl-C** — interrompe o programa que está rodando neste terminal no momento. Seu recurso padrão para encerrar um servidor de desenvolvimento ou cancelar um comando que está demorando demais.
- **Fechar a janela do terminal** também interrompe tudo que está rodando nela, mas o Ctrl-C é mais limpo.

Lembre-se do Ctrl-C e você nunca mais vai se sentir preso por um terminal “travado”.

PATH e “command not found”

Mais cedo ou mais tarde você vai digitar um comando e receber `command not found`. Isso quase nunca significa que você é burro; significa que o computador não sabe onde aquele programa mora.

O shell encontra programas verificando uma lista de pastas chamada **PATH**. Quando você digita npm, o shell percorre cada pasta do PATH procurando um programa chamado npm. Se ele não estiver em nenhuma delas, você recebe `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
```

Na prática, `command not found` geralmente significa uma de duas coisas: a ferramenta ainda não foi instalada, ou foi instalada em algum lugar que não está no seu `PATH`. Ambos os casos são comuns e têm conserto. Cole o erro para sua IA e diga o que você estava tentando rodar — resolver problemas de `PATH` é exatamente o tipo de trabalho de configuração minucioso em que uma IA é boa em te guiar.

Rodando comandos que a IA te passa – com segurança

Uma grande parte da vibe coding é a IA dizer “rode este comando” e te entregar uma linha para colar. Isso é normal e tudo bem. Mas adote um hábito que vai te proteger para sempre: **leia antes de rodar**.

Você não precisa entender cada flag. Precisa de uma noção aproximada do que o comando *faz*, especialmente do verbo no início:

- `ls`, `pwd`, `cd`, `cat`, `echo` — inofensivos. Eles olham para as coisas ou se movem por aí. Rode à vontade.
- `npm install`, `pip install` — eles baixam e configuram ferramentas. Geralmente seguros, mas alteram seu projeto.
- `rm`, `mv`, qualquer coisa com `sudo`, qualquer coisa que canaliza um download direto para o seu shell (`curl ... | bash`) — **desacelere**. Esses podem apagar coisas ou rodar código com poder total sobre sua máquina.

Se um comando parecer assustador ou você não reconhecer o verbo, pergunte: “O que este comando faz, e é seguro rodá-lo?” Uma boa IA vai explicar claramente. Nunca cole um comando de um site aleatório sem essa checagem. A linha de comando faz exatamente o que é mandada — inclusive as coisas perigosas — então o pequeno hábito de ler primeiro é o seu cinto de segurança.

Gerenciadores de pacotes que você vai encontrar

O software moderno é construído a partir de milhares de pequenas peças pré-escritas. **Gerenciadores de pacotes** são as ferramentas que baixam e organizam essas peças. Você vai encontrar dois deles o tempo todo:

```
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, Gerenciador de Pacotes do Node) é o gerenciador de pacotes para projetos JavaScript e Node. O `npm install` (frequentemente mostrado como `npm i`) lê um arquivo que lista o que o projeto precisa e baixa tudo.
- O **pip** faz o mesmo trabalho para Python.

Você vai rodar `npm install` uma vez quando abrir um projeto pela primeira vez, e depois esquecer disso. Quando a IA adicionar uma funcionalidade que precisa de um pacote novo, ela vai te dizer para instalá-lo. Agora você sabe o que isso significa: é só buscar um bloco de construção, nada a temer.

Isso é a linha de comando. Não é uma arma de hacker — é só uma caixa de texto que faz exatamente o que você pede. Aprenda a ler o prompt, reconheça uma dúzia de comandos, interrompa um processo descontrolado com Ctrl-C, e leia antes de rodar. Todo o resto, você pode perguntar.

Recapitulação e Prática

Principais aprendizados

- A linha de comando é só uma caixa de texto que faz exatamente o que você manda — palavras em vez de cliques. Um agente de IA funciona digitando os mesmos comandos que você digitaria.
- O **prompt** te diz *onde você está* toda vez; a maior parte da confusão vem de não saber sua pasta atual. `pwd`, `ls` e `cd` são 80% de tudo que você vai fazer.
- O **rm é permanente** — sem Lixeira, sem desfazer. Desacelere com `rm`, `sudo` e qualquer coisa que canaliza um download para o seu shell (`curl ... | bash`).
- Um **servidor de desenvolvimento** continua rodando e parece “travado” de propósito — isso é ele funcionando. O **Ctrl-C** interrompe qualquer programa em execução e te devolve o prompt.
- **Leia antes de rodar.** Você não precisa de cada flag, só de uma noção aproximada do verbo. Na dúvida, pergunte à IA “o que esse comando faz, e é seguro?”

Experimente

Abra seu terminal e explore sem mudar nada: rode `pwd` para ver onde você está, `ls` para ver o que há ali, entre com `cd` numa pasta e rode `ls` de novo, depois `cd ..` para subir de volta. Note como o prompt se atualiza a cada vez. Agora rode um comando que falha de propósito — digite `lss` (um erro de digitação) e leia a mensagem `command not found`. Você acabou de praticar os dois comandos mais seguros e mais usados, e aprendeu a ler um erro, tudo isso sem tocar em um único arquivo.

Prompt do 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.

Como a Web Funciona



Antes de criar algo para a web, ajuda saber o que a web realmente é. Você não precisa de um diploma de ciência da computação, e não precisa decorar nada. Você só precisa de um modelo mental funcional — uma imagem na cabeça do que acontece quando você digita um endereço e uma página aparece.

Este capítulo constrói essa imagem. Tudo que um app criado com vibe coding faz, de um formulário de login a um pagamento, é alguma versão da mesma dança simples: um computador pergunta, outro computador responde. Uma vez que você consegue ver essa dança com clareza, o resto do livro deixa de parecer mágica e passa a parecer encanamento.

Os dois personagens: cliente e servidor

Quase tudo na web é uma conversa entre dois papéis.

- **O cliente** é a coisa que *pergunta*. Na maioria das vezes é um navegador — Chrome, Safari, Firefox — rodando no seu celular ou laptop. É o cliente chegando ao balcão.

- O **servidor** é a coisa que *responde*. É um computador sentado em algum data center, rodando o dia inteiro, esperando por pedidos. É a cozinha atrás do balcão.

O cliente nunca faz o trabalho do servidor, e o servidor nunca faz o trabalho do cliente. O navegador pergunta: “Me dá a página inicial.” O servidor responde: “Aqui está.” Essa é a relação inteira. Um servidor não é uma caixa mágica especial — é só um computador comum cujo trabalho é esperar perguntas e mandar de volta respostas.

Mais uma palavra que você vai ouvir sem parar: **request** (a pergunta que o cliente envia) e **response** (a resposta que o servidor manda de volta). Guarde essas duas palavras à mão; elas são a espinha dorsal de tudo que vem a seguir.

O que uma URL realmente é

Uma URL (Uniform Resource Locator, ou Localizador Uniforme de Recursos) é o endereço que você pede. Parece uma string longa só, mas na verdade é um punhado de partes rotuladas costuradas juntas. Vamos desmontar esta aqui:

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

<code>https</code>	<code>scheme</code>	– como conversar (use HTTPS, o jeito seguro)
<code>shop.example.com</code>	<code>host</code>	– com qual servidor conversar
<code>/products/shoes</code>	<code>path</code>	– qual página ou coisa você quer dele
<code>?size=10</code>	<code>query</code>	– detalhes extras ("os do tamanho 10")

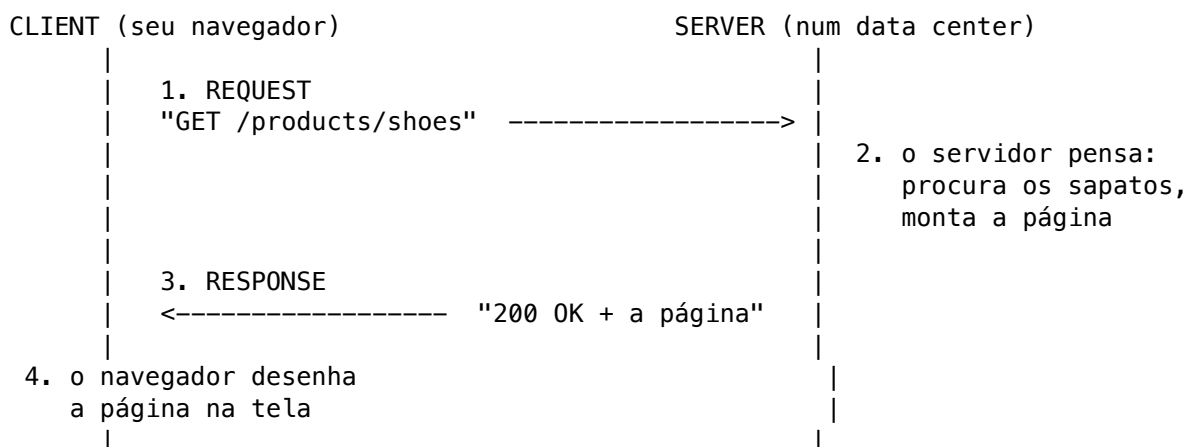
Leia como um endereço postal de trás para frente:

- O **scheme** (`https`) é o protocolo — as regras da conversa. Hoje em dia, quase sempre `https`.
- O **host** (`shop.example.com`) diz *qual* servidor contatar.
- O **path** (`/products/shoes`) diz *qual coisa* nesse servidor você quer.
- A **query** opcional (`?size=10`) carrega parâmetros extras, como filtros ou termos de busca.

Quando alguém fala “o endpoint” ou “a rota”, quase sempre está falando de um path específico em um host específico. É só isso que essas palavras assustadoras significam.

O ciclo de request e response

Aqui está a dança, do início ao fim. Você digita um endereço, e nos bastidores isso acontece em uma fração de segundo:



Quatro passos, sempre:

1. O cliente envia uma **request** para uma URL.

2. O servidor **processa** — talvez procure algo, rode alguma lógica, confira quem você é.
3. O servidor manda de volta uma **response**: um status (deu certo?) mais algum conteúdo (a página, uma imagem, alguns dados).
4. O cliente faz algo com a resposta — geralmente, desenha na sua tela.

Carregar uma página web não é uma request — são dezenas. O navegador pede a página, aí percebe que precisa de imagens, fontes e estilos, e dispara uma request nova para cada um. Mesma dança, várias vezes, bem rápido.

Métodos HTTP: os verbos

Toda request carrega um **método** — um verbo que diz que tipo de ação você quer. Existem vários, mas quatro cobrem quase tudo, e eles se encaixam direitinho em ideias do dia a dia:

- **GET** — “me dá isso.” Ler, buscar, visualizar. Carregar uma página ou uma lista. GET nunca deveria mudar nada; só recupera.
- **POST** — “aqui está algo novo.” Criar. Enviar um formulário, se cadastrar, postar um comentário.
- **PUT** — “atualiza isso.” Mudar algo que já existe, como editar seu perfil. (Você também vai ouvir **PATCH** para atualizações parciais — mesma ideia, escopo menor.)
- **DELETE** — “remove isso.” Exatamente o que parece — apagar um registro ou um arquivo.

Se você não lembrar de mais nada: **GET lê, POST cria, PUT atualiza, DELETE remove**. Quando seu app se comporta mal, saber qual verbo foi enviado costuma ser a primeira pista.

Códigos de status: deu certo?

Toda response volta com um **código de status** de três dígitos — o veredito instantâneo do servidor sobre como a request foi. Eles são agrupados pelo primeiro dígito:

- **2xx — sucesso**. Funcionou. 200 OK é o “aqui está o que você pediu” do dia a dia.
- **3xx — redirecionamento**. “Não está aqui, vá olhar ali.” Comum e inofensivo; seu navegador segue automaticamente.
- **4xx — você (o cliente) errou**. O famoso é o **404 Not Found** — você pediu um path que não existe. (Também 401/403 — você não está logado, ou não tem permissão.)
- **5xx — o servidor errou**. **500 Internal Server Error** significa que o código do próprio servidor quebrou ou travou ao processar sua request.

Vale tatuar a divisão 4xx/5xx no cérebro, porque ela diz *onde procurar*:

- Um **404** significa que a request estava errada — geralmente uma URL ruim ou um path digitado errado. Olhe o que está sendo pedido.
- Um **500** significa que o *servidor* caiu. Sua request estava certa; o código de back-end quebrou. Olhe os logs do servidor, não a URL.

Quando algo quebra num app criado com vibe coding, “é um 4xx ou um 5xx?” é uma das perguntas mais rápidas para começar a triagem.

código de status			
2xx	→	SUCESSO	funcionou (200 OK)
3xx	→	REDIRECIONAMENTO	"vai olhar ali" (seguido automaticamente)
4xx	→	erro do CLIENTE	SUA request estava errada → confira a URL/path
5xx	→	erro do SERVIDOR	o back-end quebrou → confira os logs do servidor

Headers e cookies

Junto com o conteúdo principal, toda request e response carrega **headers** — pequenas anotações rotuladas com informações extras. São como o que está escrito no envelope, separado da carta dentro. Headers dizem coisas como qual é o formato do conteúdo (Content-Type: text/html), ou quem está perguntando.

Um **cookie** é um uso específico e importante de headers. A web tem memória curta: por padrão, cada request existe sozinha e o servidor te esquece no instante em que responde. Um cookie resolve isso. Depois que você faz login, o servidor entrega ao seu navegador um pequeno token — um cookie — e seu navegador manda ele de volta automaticamente em toda request futura. É assim que um site “lembra” que você está logado entre páginas. Sem cookie, sem memória; você teria que fazer login a cada clique.

HTTPS e o cadeado

Você já viu o pequeno cadeado na barra de endereço. Ele significa que a conexão está usando **HTTPS** (HyperText Transfer Protocol Secure) — HTTP (HyperText Transfer Protocol) envolto numa camada de criptografia chamada **TLS**.

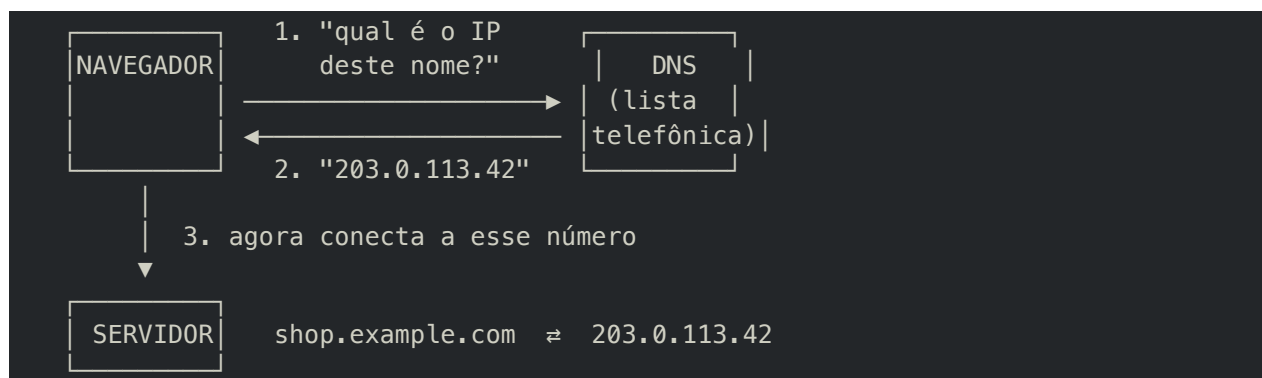
HTTP puro manda tudo como texto legível. Qualquer um sentado entre você e o servidor — num Wi-Fi compartilhado, por exemplo — poderia ler sua senha enquanto ela passa, como um cartão postal que qualquer um na cadeia de correio consegue ler. HTTPS sela a conversa num envelope que só você e o servidor conseguem abrir. O cadeado confirma duas coisas:

- **Privacidade** — ninguém no meio do caminho consegue ler o que você envia.
- **Identidade** — você está realmente conversando com o servidor que pensa estar, e não um impostor.

A regra prática é curta: apps de verdade usam HTTPS, sempre, sem exceções. Qualquer coisa que lide com logins ou pagamentos por HTTP puro está quebrada por design.

De um nome a um servidor: DNS

Uma ponta solta. Você digita `shop.example.com`, mas os computadores não se encontram de fato pelo nome — eles usam **endereços IP** numéricos como `203.0.113.42`. **DNS** (Domain Name System, ou Sistema de Nomes de Domínio) é a lista telefônica da web: antes que seu navegador possa enviar uma request, ele silenciosamente pede ao DNS para traduzir o nome de **domínio** amigável para o endereço numérico, e então se conecta a ele.



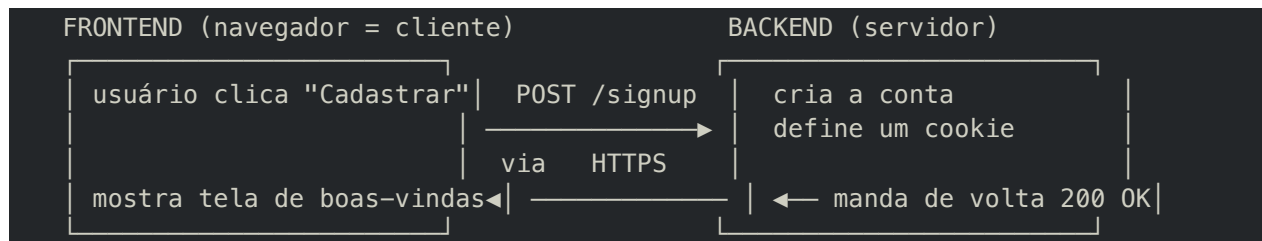
Você compra um domínio, aponta os registros de DNS dele para o servidor que roda seu app, e a partir daí o nome leva as pessoas até sua máquina. É um passo de busca que você quase nunca vai ver, mas é por isso que um nome fácil de lembrar consegue substituir uma sequência de números.

Como isso se encaixa no seu app

Agora sobreponha o quadro inteiro a um app criado com vibe coding, porque cada parte que você vai construir vive de um lado da dança:

- O **frontend** é tudo que roda no navegador — os botões, formulários e layout que o usuário vê e clica. É o cliente. Ele majoritariamente envia requests e exibe responses.
- O **backend** é seu código rodando num servidor — a lógica que decide o que fazer com uma request, conversa com o banco de dados, e monta a response. É o servidor.

Aqui está toda a jornada de cadastro numa imagem só — cada palavra em negrito do capítulo sentada onde pertence:



Uma interação real costura tudo isso junto. Um usuário clica em “Cadastrar” (frontend). O navegador manda uma request **POST** via **HTTPS** para um path como **/api/signup**, carregando os dados do formulário. Seu backend a recebe, cria a conta, define um **cookie** para o usuário continuar logado, e manda de volta um **200**. O frontend vê o sucesso e mostra uma tela de boas-vindas. Toda palavra em negrito ali é algo que você acabou de aprender — e essa é a forma inteira de quase tudo que você vai construir.

Você não precisa guardar tudo isso de cabeça de uma vez. Mas quando um capítulo mais à frente disser “a request chegou no servidor mas voltou 500”, ou “define este header”, ou “esta rota processa o POST”, você agora vai saber exatamente para qual parte da dança isso aponta. Esse modelo mental é o verdadeiro pré-requisito — e você já tem ele.

Recapitulando e Praticando

Principais aprendizados

- A web é uma dança simples repetida infinitamente: um **cliente** (geralmente um navegador) manda uma **request**, e um **servidor** manda de volta uma **response**. Todo o resto é detalhe em cima disso.
- Uma **URL** é partes rotuladas costuradas juntas — scheme, host, path, query. O método é o verbo: **GET** lê, **POST** cria, **PUT** atualiza, **DELETE** remove.
- **Códigos de status** são o veredito instantâneo do servidor. A divisão 4xx/5xx diz onde procurar: **4xx** significa que a request estava errada (URL ruim), **5xx** significa que o próprio código do servidor quebrou.
- **HTTPS** (o cadeado) sela a conversa para que ninguém no meio consiga ler — apps de verdade usam sempre. **Cookies** são como um site lembra que você está logado entre requests.
- Seu **frontend** é o cliente (roda no navegador); seu **backend** é o servidor (roda sua lógica e conversa com o banco de dados). Toda funcionalidade que você constrói vive de um lado da dança.

Experimente

Abra qualquer site, depois abra as ferramentas de desenvolvedor do seu navegador (clique com o botão direito → “Inspecionar”, depois a aba “Network”) e recarregue a página. Observe a lista de requests se enchendo — você vai ver uma request de página seguida de dezenas mais para imagens, fontes e estilos, exatamente como descrito. Clique em uma e olhe seu **método** (GET), seu **código de status** (esperançosamente 200), e seus **headers**. Agora você está vendo a dança de request/response acontecer em tempo real, com o vocabulário para lê-la.

Prompt do capítulo

Estou aprendendo como a web funciona como iniciante. Usando meu próprio app como exemplo: <descreva seu app, ou cole uma URL ou um erro que você está vendo>

- Me explique o ciclo de request/response para uma ação específica (como carregar uma página ou enviar um formulário), passo a passo.
- Me diga qual parte é o cliente (frontend) e qual é o servidor (backend).
- Se eu estiver vendo um código de status ou erro, explique se é um 4xx (minha request estava errada) ou um 5xx (o servidor quebrou), e onde procurar.
- Mantenha concreto e amigável para iniciantes – sem conhecimento prévio assumido.

APIs, JSON e Dados



Praticamente nenhum aplicativo que você construir vai viver sozinho. No momento em que você quer mostrar um mapa, cobrar um cartão, enviar um e-mail ou consultar a previsão do tempo, seu app precisa *conversar com o aplicativo de outra pessoa*. A linguagem que ele fala se chama **API** (Application Programming Interface, ou Interface de Programação de Aplicações), e as palavras que vão e voltam quase sempre são escritas em um formato chamado **JSON** (JavaScript Object Notation). Este capítulo fala sobre os dois — não para que você escreva a “encanamento” na mão, mas para que você consiga ler, conferir o que a IA construiu, e perceber quando algo está errado antes que isso custe dinheiro.

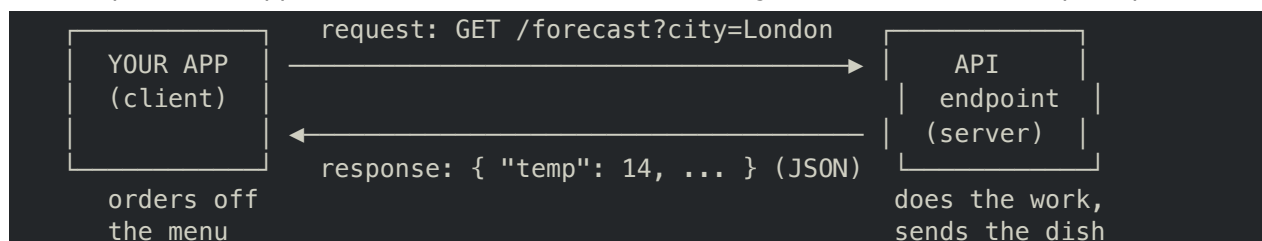
Você não precisa decorar nada aqui. Você precisa reconhecer as formas. Quando a IA disser “vou chamar a API da Stripe e processar a resposta em JSON”, você deve conseguir imaginar mais ou menos o que isso significa e o que pode dar errado.

O que uma API realmente é

A forma mais clara de pensar em uma API é como um **cardápio de pedidos que um programa oferece a outro**.

Um restaurante não deixa você entrar na cozinha e cozinhar. Ele te entrega um cardápio: aqui estão as coisas que você pode pedir, aqui está como pedir, aqui está o que você vai receber de volta. Você não precisa saber como a cozinha funciona. Você só precisa do cardápio. Uma API é exatamente isso — uma empresa diz “aqui estão os pedidos que você pode fazer ao nosso serviço, e aqui está o que cada um retorna.”

Quando a IA escreve código que “chama a API do tempo”, ela está fazendo um pedido daquele cardápio: *me dê a previsão para esta cidade*. Os servidores da empresa de meteorologia fazem o trabalho e devolvem uma resposta. Seu app nunca vê o banco de dados ou o código deles. Ele só recebe o prato pronto.



A virada de mentalidade importante: uma API é um **contrato**. A outra empresa promete que, se você perguntar de um jeito específico, vai receber uma resposta em um formato específico. Seu trabalho — e o da IA — é perguntar corretamente e ler a resposta corretamente. A maioria dos bugs de API é um dos lados quebrando esse contrato.

Endpoints e REST, em termos simples

Um **endpoint** é um único item do cardápio. É apenas uma URL (Uniform Resource Locator, ou Localizador Uniforme de Recursos) que faz uma coisa específica. Por exemplo:

- `https://api.weather.com/forecast` — pegar uma previsão
- `https://api.weather.com/cities` — listar cidades disponíveis

Mesma cozinha, pratos diferentes. Cada endpoint é uma capacidade.

A maioria das APIs que você vai usar segue um estilo informal chamado **REST** (Representational State Transfer, ou Transferência de Estado Representacional). Você não precisa da definição acadêmica; precisa de duas ideias:

- **A URL nomeia uma coisa** (um “recurso”) — um usuário, um pedido, uma previsão.
- **O verbo diz o que você quer fazer com ela**. Esses verbos são métodos HTTP (HyperText Transfer Protocol, ou Protocolo de Transferência de Hipertexto), os mesmos do capítulo sobre como a web funciona:
 - GET — ler algo (buscar uma previsão). Seguro; não muda nada.
 - POST — criar algo (fazer um pedido, cadastrar um usuário).
 - PUT / PATCH — atualizar algo que já existe.
 - DELETE — remover algo.

Então `GET /orders/42` significa “ler o pedido número 42,” e `DELETE /orders/42` significa “apagá-lo.” Mesmo substantivo, verbo diferente, efeito completamente diferente. Quando você lê a documentação de uma API ou um diff, essas duas palavras — o **método** e a **URL** — dizem a maior parte do que uma requisição faz.

O que é JSON

Quando seu app faz uma pergunta a uma API, a resposta volta como texto em um formato chamado **JSON**. Ele se tornou o padrão porque é legível por humanos e fácil de os programas manipularem. Você vai vê-lo o tempo todo, então vale a pena saber ler.

JSON é construído a partir de **chaves e valores** (keys e values). Uma chave é um rótulo entre aspas; um valor é o dado associado a ela. Aqui está um exemplo real descrevendo um usuário:

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

Leia como um formulário que alguém preencheu. "name" é um campo, "Ada Lovelace" é o que a pessoa escreveu nele. Duas estruturas fazem todo o trabalho:

- Um **objeto** é envolvido por chaves { }. É um pacote de pares chave/valor — como um único registro. Todo o exemplo acima é um objeto.
- Um **array** é envolvido por colchetes []. É uma lista ordenada de valores. Acima, "tags" contém um array com duas strings.

Repare que "profile" é um objeto *dentro* do objeto. JSON permite aninhamento — um valor pode ser, ele mesmo, um objeto ou um array, tão profundo quanto necessário. É assim que uma única resposta pode descrever um usuário, o endereço dele e seus últimos dez pedidos, tudo de uma vez.

Desenhado como uma árvore, o mesmo objeto de usuário mostra sua forma de relance — campos simples no topo, uma lista e um objeto aninhado se ramificando:

```
{ } 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)
```

Os tipos de dados que você vai encontrar

Todo valor em JSON pertence a um pequeno conjunto de tipos. Conhecê-los te permite perceber quando um valor está do tipo errado — uma fonte frequente de bugs.

- **string** — texto, sempre entre aspas duplas: "Ada Lovelace", "ada@example.com".
- **number** — um número simples, sem aspas: 42, 3.14. (As aspas mudam o significado: "42" é o *texto* quatro-dois, não o número.)
- **boolean** — true ou false, sem aspas. Usado para indicadores de sim/não como "isAdmin".

- **null** — um deliberado “não há valor aqui.” `"avatarUrl": null` significa que o usuário não tem avatar — diferente da chave estar simplesmente ausente.
- **array** — uma lista ordenada: `["beta", "early-access"]`.
- **object** — um pacote de pares chave/valor: o bloco `"profile"`.

Esse é todo o vocabulário. O erro mais comum na prática é a armadilha string-versus-number: o código espera o número 42, mas a API envia a string "42", e uma comparação falha silenciosamente. Quando algo se comporta de forma estranha, checar o *tipo* de um valor costuma ser a solução.

Uma requisição e resposta reais

Aqui está a aparência de uma chamada de API real, do começo ao fim — a requisição que seu app envia, e o JSON que ele recebe de volta:

```
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 }
  ]
}
```

Aquela linha `Authorization` é o próximo assunto — e o que mais vale a pena acertar.

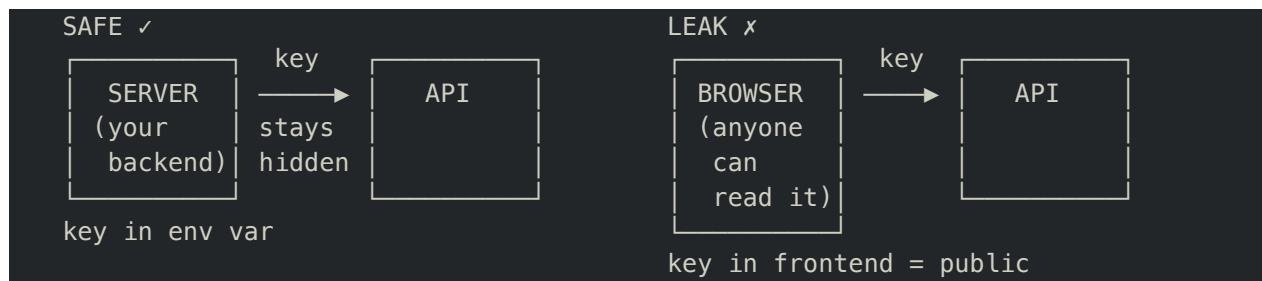
Chaves de API e cabeçalhos de autenticação

A maioria das APIs úteis não é aberta ao mundo. A empresa precisa saber *quem* está perguntando, para poder rastrear o uso e cobrar por ele. Ela te dá uma **chave de API** (API key) — uma longa string secreta que identifica sua conta. Você a anexa a toda requisição, geralmente em um **cabeçalho HTTP** (HTTP header) chamado `Authorization`, como na linha `Bearer sk_live_...` acima.

Trate uma chave de API exatamente como uma senha, porque é isso que ela é. Qualquer um que a tenha pode fazer requisições *como se fosse você* — e estourar sua fatura ou ler seus dados. Isso se conecta diretamente ao capítulo de segurança, e há uma regra que importa mais do que as outras:

- **Nunca coloque uma chave de API em código que vai para o navegador.** Tudo o que está no seu frontend é público — usuários podem abrir as ferramentas de desenvolvedor e lê-lo. A IA, ao ser pedida para “chamar a API a partir da página”, vai alegremente colar sua chave secreta bem ali. Impeça isso. Chaves secretas pertencem ao *servidor*.
- **Nunca faça commit de uma chave no git.** Mesmo que você a apague depois, ela permanece para sempre no histórico, e robôs vasculham repositórios públicos atrás de chaves em minutos.
- **Guarde chaves em variáveis de ambiente**, não no próprio código — o mesmo padrão coberto no capítulo de segurança.

A chave precisa viajar a partir de um lugar seguro, nunca a partir do navegador. Imagine os dois caminhos — um seguro, um vazamento:



Se você algum dia vir uma chave real sentada em um diff, em um arquivo de frontend, ou em uma mensagem de chat, trate-a como já vazada: rotacione-a (gere uma nova, desative a antiga) imediatamente.

Limites de taxa e a armadilha da conta chocante

Mais duas coisas que o cardápio não anuncia mas que vão definitivamente te morder.

Limites de taxa (rate limits). APIs limitam quantas requisições você pode fazer em uma janela de tempo — digamos, 100 por minuto. Ultrapasse e você é rejeitado com um erro `429 Too Many Requests` até desacelerar. Isso existe para impedir que um cliente sobrecarregue o serviço. A armadilha: código escrito por IA frequentemente chama uma API dentro de um laço (loop) sem nenhuma pausa, estourando o limite instantaneamente. Se uma funcionalidade funciona para um item mas quebra quando você a executa mil vezes, um limite de taxa é o principal suspeito.

APIs medidas e pagas. Muitas APIs cobram por requisição — frações de centavo cada. Isso não é nada para algumas chamadas e vira uma conta de verdade quando algo entra em loop descontrolado. A história de terror clássica: um bug tenta novamente uma chamada paga que falhou para sempre, ou um formulário público deixa estranhos disparar chamadas pagas, e você acorda com uma fatura de quatro dígitos. Proteja-se:

- Leia o preço *antes* de conectar uma API paga. Saiba o custo por chamada.
- Configure um **teto de gastos ou alerta de orçamento** no painel do provedor, se ele oferecer isso. A maioria oferece.
- Desconfie de qualquer código que chame uma API paga dentro de um laço ou a cada carregamento de página.

Como ler a documentação de uma API

Você vai viver dentro de documentações de API, então aqui está como conseguir o que precisa rapidamente sem ler tudo. Para qualquer endpoint, procure por cinco coisas:

- **O método e a URL** — `GET /forecast`. O que ele faz, em dois tokens.
- **Autenticação** — que cabeçalho ou chave ele exige. Geralmente uma seção compartilhada para toda a API.
- **Parâmetros** — as entradas que você pode enviar (aqui, `city`), quais são obrigatórias, e seus tipos.
- **Um exemplo de resposta** — o formato de JSON que você vai receber de volta. Isso é ouro; copie-o.
- **Erros** — o que volta quando algo dá errado, e os códigos de status (`401` não autorizado, `404` não encontrado, `429` limite de taxa atingido).

Boas documentações incluem um exemplo de requisição e resposta prontos para copiar e colar. Esses dois trechos dizem mais do que parágrafos de texto — pegue-os e entregue-os à IA.

Deixando a IA fazer as chamadas — enquanto você confere o formato

Aqui está a divisão prática de trabalho. A IA é genuinamente boa em escrever código que chama APIs: ela conhece as bibliotecas, a sintaxe dos cabeçalhos, o tratamento de erros. Deixe que ela faça isso. Você

não precisa escrever esse encaimento na mão.

Mas você é dono de uma coisa em que a IA é ruim: **saber se o formato dos dados está realmente correto**. Um bom fluxo de trabalho:

- **Dê à IA a documentação real.** Cole o exemplo de requisição e resposta da documentação no seu prompt. A IA chutando o formato de uma API de memória é uma das principais fontes de bugs — ela vai inventar chaves que parecem plausíveis mas não existem. Exemplos reais a ancoram na realidade.
- **Peça para ela te mostrar a resposta bruta primeiro.** “Antes de processar, registre (log) o JSON real que a API retorna.” Depois, compare visualmente com o que o código espera. A chave que ele lê (`response.current.temp`) realmente existe na resposta? Metade dos bugs de API é um nome de chave que não bate.
- **Confira os tipos.** Isso é um número ou uma string? Um campo que deveria sempre estar presente às vezes vem `null`? Você já sabe ler JSON — use isso.
- **Trate o caminho infeliz.** Pergunte: “O que acontece se essa chamada falhar, expirar (timeout), ou atingir o limite de taxa?” Se a resposta for “o app trava,” isso é um bug, não uma funcionalidade.

Você não precisa escrever a requisição. Você precisa ler a resposta e perguntar “esse é o formato que eu esperava?” Esse único hábito — checar os dados contra o contrato — pega a maioria dos bugs de integração antes que seus usuários sequer os vejam.

Recapitulando e Praticando

Principais pontos

- Uma **API** é um cardápio de pedidos que um programa oferece a outro, e é um **contrato**: pergunte da forma combinada, receba uma resposta no formato combinado. A maioria dos bugs de API é um dos lados quebrando esse contrato.
- **JSON** é como a resposta volta — chaves e valores, construídos a partir de objetos `{ }` e arrays `[ ]`, que se aninham. Os tipos de dados são `string`, `number`, `boolean`, `null`, `array`, `object`.
- A **armadilha string-versus-number** (42 vs “42”) e **ausente-versus-null** são os bugs de formato mais comuns. Você já sabe ler JSON agora — confira o tipo quando algo se comportar de forma estranha.
- Uma **chave de API** é uma senha. Nunca a envie para o navegador, nunca a comite no git, guarde-a em variáveis de ambiente. Uma chave vazada deve ser rotacionada imediatamente.
- Fique atento a **limites de taxa** (429) e **cobrança medida** — laços escritos por IA que chamam uma API paga sem pausa são como você acaba com uma fatura de quatro dígitos. Leia o preço e configure um alerta de orçamento antes.

Experimente

Encontre uma API gratuita e sem chave no seu navegador — por exemplo, abra <https://api.github.com/users/octocat> diretamente em uma nova aba. Você vai ver uma resposta JSON bruta. Leia-a como um formulário: identifique os objetos `{ }`, encontre um array se houver algum, e nomeie o tipo de três valores diferentes (quais são strings? numbers? booleans? algo é `null`?). Você acabou de fazer exatamente a checagem que este capítulo pede para você fazer em toda resposta de API que a IA conectar.

Prompt do 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.
```

Anatomia de um App



Quando você pede a uma IA para “adicionar uma página de login” ou “salvar isso no banco de dados”, ela está silenciosamente tomando decisões sobre meia dúzia de peças móveis. Se você não sabe que essas peças existem, as respostas soam como uma língua estrangeira — e você não consegue distinguir uma boa resposta de uma confusa.

Este capítulo é um mapa. Ele não vai te transformar em engenheiro, mas vai te permitir apontar para qualquer parte de um app moderno e dizer o que ela faz e onde ela vive. Com o mapa em mãos, todos os outros capítulos deixam de ser abstratos: você vai saber *em qual* caixa um problema mora antes mesmo de começar a descrevê-lo. Vamos manter tudo simples — nenhum jargão sem definição, e um único exemplo recorrente (um usuário clicando em um botão) rastreado através de cada camada.

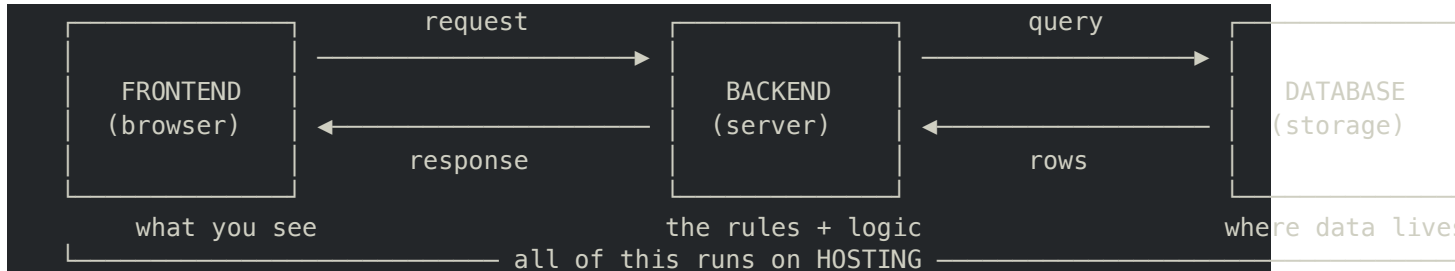
O panorama geral

Praticamente todo app que você vai construir é feito de quatro partes que conversam entre si:

- **Frontend** — o que o usuário vê e clica. As telas, botões e textos dentro de um navegador ou de um aplicativo de celular.

- **Backend** — a lógica que roda em um servidor que você controla. Ele aplica as regras, faz o trabalho pesado e decide quem pode fazer o quê.
- **Banco de dados** — onde a informação vive para continuar lá amanhã. Contas, posts, pedidos.
- **Hospedagem (hosting)** — os computadores, em algum lugar do mundo, que de fato executam tudo isso para que outras pessoas consigam alcançar o app.

Veja como eles se conectam:



O frontend nunca conversa diretamente com o banco de dados. Ele sempre passa pelo backend, onde as regras vivem. Esse único fato explica boa parte de como os apps são construídos — e por que “simplesmente deixar a página ler o banco de dados” quase nunca é a resposta certa.

Frontend, backend, banco de dados, hospedagem

Vamos descer mais um nível, usando nosso exemplo: um usuário clica em “**Salvar perfil.**”

Frontend. O app visível, rodando em um navegador web ou como um aplicativo de celular. Ele desenha o formulário, percebe o clique e coleta o que o usuário digitou. Por design, ele é *não confiável*: qualquer pessoa pode abri-lo, inspecioná-lo e alterá-lo. Então o trabalho dele é parecer bom e responder rápido, não aplicar nada que realmente importe.

Backend. Quando o botão é clicado, o frontend envia os dados para o backend — código rodando em um servidor. O backend verifica se o usuário está logado, valida os dados (“isso é realmente um endereço de e-mail?”), aplica as regras de negócio e só então salva. É aqui que a confiança mora, porque os usuários não conseguem interferir nisso.

Banco de dados. O backend entrega os dados ao banco de dados para armazená-los. Não é uma caixa mágica — pense nele como um conjunto de planilhas muito rígidas (chamadas *tabelas*) que podem ser pesquisadas instantaneamente e nunca esquecem nada. Amanhã, o backend lê o perfil de volta *do* banco para exibi-lo novamente.

Hospedagem. Em produção, nada disso roda no seu notebook. Roda em computadores alugados de um provedor de hospedagem como Cloudflare, Vercel, ou uma plataforma de nuvem. Hospedagem é o que torna seu app acessível em um endereço real na web, em vez de existir apenas na sua própria máquina.

Um instinto útil: quando algo quebra, pergunte *em qual camada*. “O botão não faz nada” é frontend. “Ele salva a coisa errada” geralmente é backend. “Ele esqueceu meus dados” aponta para o banco de dados. Nomear a camada é metade do caminho para consertar o bug — e metade do caminho para escrever um prompt sobre o qual a IA consiga agir.

HTML, CSS e JavaScript

O frontend, dentro de um navegador, é construído a partir de exatamente três linguagens. Você não precisa escrevê-las à mão, mas precisa saber o que cada uma é *responsável por*.

HTML (HyperText Markup Language, ou Linguagem de Marcação de Hipertexto) é a estrutura — o conteúdo e seu esqueleto. Ele diz “aqui tem um título, aqui tem um parágrafo, aqui tem um botão.” São os substantivos da página; sozinho, o HTML é um documento simples e sem estilo, como um arquivo de

texto com etiquetas. O navegador transforma seu HTML em uma árvore viva de objetos em memória chamada **DOM** (Document Object Model, ou Modelo de Objeto de Documento); quando um código “muda a página”, na verdade ele está mudando o DOM, e o navegador redesenha a tela para acompanhar.

CSS (Cascading Style Sheets, ou Folhas de Estilo em Cascata) é o estilo — a aparência de tudo. Cores, fontes, espaçamento, layout, alinhado à esquerda ou centralizado, tela pequena versus tela grande. O CSS não muda o *que* está na página, apenas como isso é apresentado. Se o HTML são os substantivos, o CSS são os adjetivos.

JavaScript (JS) é o comportamento — o que acontece quando você interage. Ele fica escutando o clique, envia os dados para o backend, mostra um indicador de carregamento, atualiza o DOM quando a resposta chega. HTML e CSS são estáticos; o **JavaScript** faz uma página *fazer* coisas. São os verbos. (A mesma linguagem também costuma rodar o backend, por isso você a vê em todo lugar.)

Uma forma de visualizar essa divisão: o HTML constrói uma árvore (o DOM), o CSS a pinta, e o JavaScript entra e a modifica:



Pacotes e dependências

Ninguém escreve um app inteiro do zero. Os problemas difíceis e comuns — formatar datas, processar pagamentos, exibir um calendário — já foram resolvidos e empacotados por outras pessoas. Um **pacote** é um pedaço reutilizável de código que você importa para o seu projeto em vez de escrever você mesmo. As **dependências** do seu projeto são a lista de pacotes dos quais ele depende.

No mundo do JavaScript, a ferramenta que gerencia isso é o **npm** (Node Package Manager, ou Gerenciador de Pacotes do Node). Dois arquivos fazem essa contabilidade:

- **package.json** — a lista legível por humanos do que seu projeto precisa, além dos comandos para executá-lo. Você (ou a IA) edita este arquivo.
- **o lockfile** (package-lock.json ou similar) — um registro exato, gerado por máquina, de cada pacote e sua versão precisa, para que o app seja construído de forma idêntica na sua máquina, na de um colega e no servidor. Você não edita isso manualmente; deixa a ferramenta gerenciar.

```
{
  "name": "my-app",
  "scripts": {
    "dev": "next dev",
    "build": "next build"
  },
  "dependencies": {
    "next": "16.0.0",
    "react": "19.0.0"
  }
}
```

Por que isso importa? Quando a IA diz “vamos adicionar uma biblioteca para isso”, ela está editando o package.json e você está confiando no código de outra pessoa — poucas dependências bem conheci-

das são mais seguras do que uma pilha de dependências obscuras. E quando um build quebra misteriosamente, uma dependência incompatível é uma suspeita frequente, e “apagar e reinstalar os pacotes” é uma solução real e comum.

Variáveis de ambiente e configuração

Alguns valores não podem ser escritos dentro do seu código: chaves secretas, senhas de banco de dados, o endereço de um serviço. E alguns precisam *mudar* dependendo de onde o app está rodando. O código que você testa no seu notebook (o ambiente de **dev**, ou desenvolvimento) deve conversar com um banco de dados de teste; o app ao vivo que seus usuários acessam (o ambiente de **prod**, ou produção) precisa conversar com o banco real — usando o *mesmo código*.

A resposta são as **variáveis de ambiente** (frequentemente chamadas de “env vars”): valores nomeados que vivem *fora* do código e são fornecidos a ele quando é executado. O código diz “me dê DATABASE_URL,” e o ambiente decide qual valor entregar.

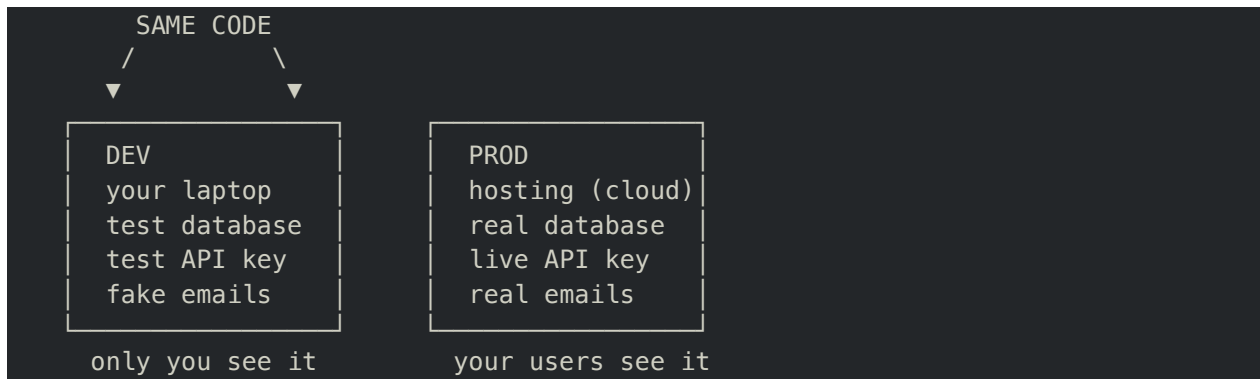
```
# A config file (e.g. .env) – values live here, not in the code

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
```

Duas regras vão te poupar de dores de cabeça reais:

- **Segredos nunca vão dentro do código.** Qualquer coisa no seu código pode acabar se tornando pública (especialmente em um repositório compartilhado), e uma chave de pagamento ou senha de banco de dados vazada é uma emergência de verdade. Segredos pertencem às variáveis de ambiente, mantidos fora do controle de versão.
- **Dev e prod são diferentes.** O ponto é que o *mesmo* código se comporta corretamente em ambos, porque o ambiente — não o código — decide qual banco de dados e quais chaves usar.

O mesmo código, duas casas — as variáveis de ambiente são o que muda:



A forma de um projeto

Abra um projeto típico e a lista de pastas pode parecer um monte de ruído. Não é — a maioria dos projetos segue um padrão reconhecível. Aqui está um app web simplificado:

```
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

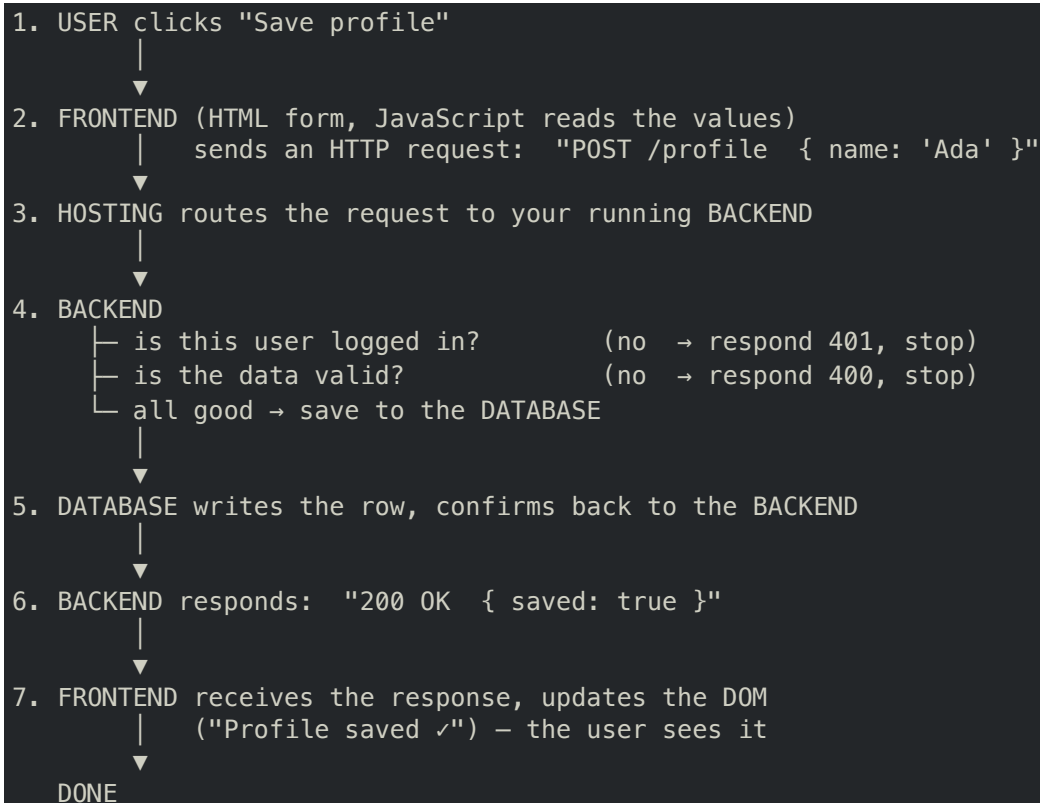
```

Você não precisa decorar isso — só precisa reconhecer, para que quando a IA disser “vou adicionar isso em `src/lib`,” você saiba mais ou menos onde fica isso e por quê. Algumas convenções fundamentais:

- **src/** guarda o código que você escreve. A bagunça no nível superior é, na maior parte, configuração.
- **Arquivos de configuração na raiz** (o `package.json`, os dotfiles) configuram as ferramentas, não o comportamento do seu app.
- **.gitignore** lista os arquivos que *não* devem ser salvos no controle de versão — seu `.env` pertence aqui, o que é exatamente como os segredos ficam fora de um repositório compartilhado.

Como uma requisição flui

Agora vamos amarrar tudo. Vamos acompanhar nosso usuário clicando em “**Salvar perfil**” através de cada peça que você acabou de conhecer. Este é o modelo mental mais útil de todo o livro; se você internalizar apenas um diagrama, que seja este.



Percorra o que cada camada fez: o **frontend** coletou a entrada e a enviou; a **hospedagem** levou a requisição até o lugar certo; o **backend** aplicou as regras (login, validação) e decidiu; o **banco de dados** tornou tudo permanente; e a resposta fluiu de volta até que o **frontend** pudesse atualizar a tela. Todo app é uma variação desse ciclo, executado milhares de vezes por segundo.

Repare que todas as verificações acontecem no backend, nunca no frontend — a regra da primeira seção, em ação: o frontend é para o usuário, o backend é para a verdade. Quando você sabe *onde* cada etapa

acontece, você consegue descrever uma funcionalidade com precisão (“valide o e-mail no backend antes de salvar”) em vez de vagamente (“faça o formulário funcionar”) — e precisão é o jogo inteiro.

Recapitulação e Prática

Principais aprendizados

- Todo app tem quatro partes: um **frontend** (o que os usuários veem), um **backend** (as regras e a lógica), um **banco de dados** (onde os dados persistem) e uma **hospedagem** (onde tudo isso roda). O frontend nunca toca o banco de dados diretamente — ele passa pelo backend.
- No navegador, **HTML** é estrutura, **CSS** é estilo, e **JavaScript** é comportamento; o código muda a página ao mudar o **DOM**.
- Você não escreve tudo sozinho — **pacotes** (gerenciados pelo **npm** via `package.json` e um lockfile) fornecem problemas já resolvidos como **dependências**.
- **Variáveis de ambiente** mantêm segredos fora do código e permitem que o *mesmo* código se comporte corretamente em dev e prod.
- Uma requisição flui frontend → hospedagem → backend → banco de dados e de volta; nomear a camada onde um problema mora é metade do caminho para resolvê-lo.

Coloque em prática

Escolha um app que você usa todo dia e narre em voz alta uma ação através das camadas: “Eu toco em *enviar* (frontend) → isso vai para o servidor deles (backend) → que verifica se eu tenho permissão e salva a mensagem (banco de dados) → e o chat é atualizado (frontend).” Depois, tente adivinhar em qual camada mora cada problema: o botão parece desalinhado; o app diz “você não está logado”; uma mensagem de ontem sumiu. Ficar bom nessa nomeação é a habilidade sobre a qual o resto do livro é construído.

Prompt do 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.
```

A mentalidade do vibe coding



Vibe coding é um termo carregado de significados. Para alguns, significa digitar um prompt de uma linha, aceitar o que quer que o modelo devolva, e publicar sem ler uma única linha. Essa versão é um brinquedo. Ela produz demos que desmoronam no instante em que um usuário real as toca.

Este livro é sobre a outra versão: direcionar uma IA para construir software de produção da mesma forma que um engenheiro sênior direciona uma equipe. Você continua no comando do *o quê* está sendo construído e do *porquê* está correto. Você delega a digitação. Essa é a mudança, e ela é maior do que parece.

O que é vibe coding, de fato

Vibe coding é **construir software expressando intenção em linguagem natural e deixando uma IA gerar a implementação**, enquanto você direciona, revisa e refina em um ciclo apertado.

Não é:

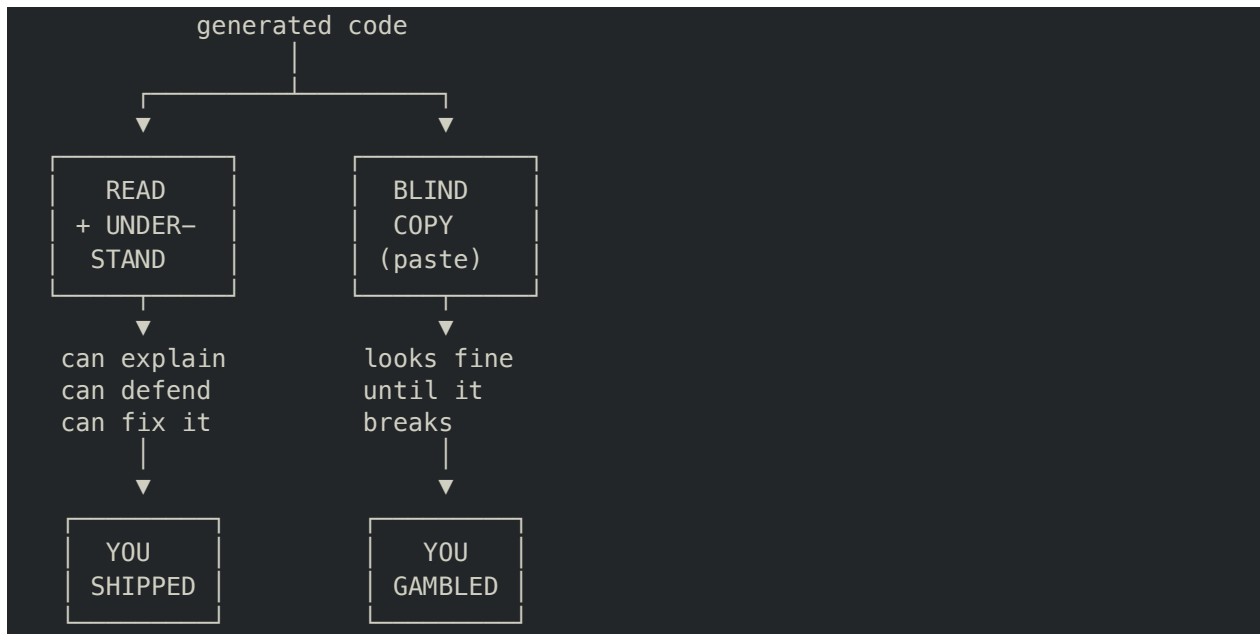
- Aceitar cegamente código gerado que você não entende.
- Substituir o julgamento de engenharia por “vibe” (sensação, instinto).
- Um botão mágico que transforma uma frase em um produto lançado.

É:

- Tratar o modelo como um engenheiro júnior rápido, conhecedor e um pouco confiante demais.
- Gastar sua atenção em arquitetura, correção e bom senso em vez de sintaxe.
- Iterar em minutos sobre coisas que antes levavam horas.

O modelo mental que funciona melhor: você é o tech lead (líder técnico), a IA é sua equipe. Um bom tech lead não escreve cada linha. Ele decide a direção, define restrições, revisa o trabalho que sua equipe envia para aprovação (no software, esse lote de mudanças proposto se chama pull request) e detecta as más ideias antes que sejam publicadas. Você está fazendo isso — só que na velocidade de uma conversa.

Esses dois caminhos partem do mesmo código gerado e terminam em lugares muito diferentes:



Uma palavra nessa definição carrega todo o peso: **entender**. Toda a abordagem repousa sobre uma única regra — *entenda o que você publica*. Você não precisa ter digitado, mas precisa ser capaz de explicar, defender em uma revisão e consertar às 2 da manhã quando o sistema te chamar por causa de um alerta. Se você não consegue, você não fez vibe coding; você apostou e teve sorte. A diferença entre essas duas coisas é o assunto inteiro deste livro.

A mudança de mentalidade

Se você veio de digitar cada caractere você mesmo, a parte mais difícil não é aprender as ferramentas. É soltar o teclado.

Seu antigo gargalo era a *produção* — quão rápido você conseguia escrever código correto. Seu novo gargalo é a *especificação* e a *revisão* — quão claramente você consegue descrever o que quer e quão bem você consegue julgar o que volta.

Isso muda o que você otimiza:

- **Antes:** memorizar APIs (Application Programming Interfaces — interfaces de programação de aplicações, os “cardápios” de comandos que uma ferramenta expõe), lutar com sintaxe, digitar rápido.
- **Agora:** descrever a intenção com precisão, decompor problemas, avaliar o resultado de forma crítica.

Os engenheiros que prosperam nisso não são os digitadores mais rápidos. São aqueles com opiniões fortes sobre como é “bom” e a disciplina de rejeitar resultados que não atingem esse padrão.

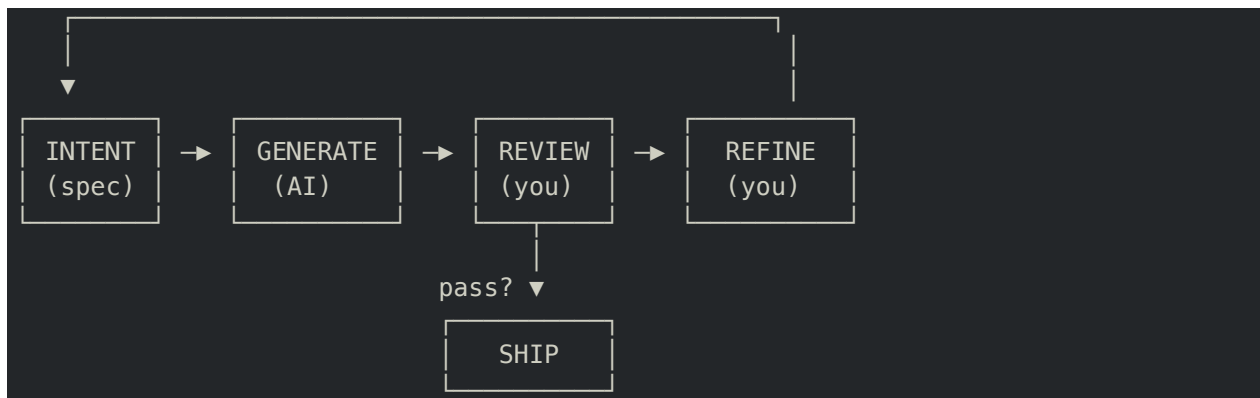
Há uma segunda mudança, mais silenciosa, que pega as pessoas de surpresa: sua relação com *não saber*. O instinto antigo era parar e aprender a API antes de tocá-la. Agora você muitas vezes consegue gerar uma primeira versão em uma stack desconhecida e aprender lendo o resultado. Isso é um superpoder real — e uma armadilha. Ler código que você não conseguiria ter escrito é normal. Publicar código que você não consegue *avaliar* não é. A linha divisória é se você consegue afirmar, com confiança, que o resultado

está correto. Apoie-se no modelo para escrever mais rápido do que você conseguiria; nunca se apoie nele para saber mais do que você está disposto a verificar.

O ciclo: intenção → gerar → revisar → refinar

Tudo neste livro volta a um único ciclo. Internalize-o.

O ciclo gira até que o resultado atinja o seu padrão — e a revisão é o portão pelo qual ele precisa passar toda vez:



1. **Intenção.** Declare o que você quer e as restrições que importam — entradas, saídas, casos de borda, a stack, o estilo. Intenção vaga gera código vago.
2. **Gerar.** Deixe o modelo escrever a implementação. Não fique guiando a sintaxe passo a passo; descreva o comportamento.
3. **Revisar.** Leia o que voltou. Ele realmente faz o que deveria? Ele trata a lista vazia, o nulo, a chamada de rede que falha? É a versão *mais simples* que funciona?
4. **Refinar.** Aponte o que está errado, dê uma correção concreta e gere novamente. Repita até atingir o seu padrão.

A maioria dos iniciantes pula o passo 3. Eles geram, roda, e seguem em frente. Aí quebra em produção no caso que nunca foi descrito. O passo de revisão é onde a engenharia acontece de fato.

Veja como é um bom primeiro prompt nesse 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.
```

Repare na estrutura: uma assinatura precisa, regras explícitas, os casos de borda apontados, as restrições declaradas e um pedido de testes. Você não está torcendo para que o modelo acerte no chute — você está eliminando a ambiguidade que produz chutes ruins.

O passo de refinar é uma habilidade tão importante quanto o primeiro prompt. Feedback vago gera correções vagas. Compare estas duas correções:

```
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'."
```

A boa versão nomeia o sintoma, a causa e a correção. Você está revisando como faria com o pull request de um colega — e quanto mais preciso for o seu apontamento, menos rodadas você gasta. Cada ciclo deve *convergir*. Se você se pegar girando em círculos, esse é um sinal de que sua intenção estava subespecificada; pare de gerar de novo e reescreva a especificação em vez disso.

Onde brilha

O vibe coding é genuinamente transformador para:

- **Estruturação do zero (greenfield).** Subir um app novo, ligar rotas, configurar o ambiente — trabalho tedioso, mas bem conhecido.
- **Boilerplate e cola.** Endpoints CRUD, validação de formulários, transformações de dados, adaptadores entre duas APIs.
- **Território desconhecido.** Uma linguagem ou framework que você não domina bem. O modelo conhece as convenções; você fornece o julgamento.
- **Refatorações e migrações.** “Converta essas promises para async/await”, “divida esse arquivo de 400 linhas por responsabilidade”.
- **Exploração descartável.** Protótipos que você vai reescrever de qualquer forma. Velocidade importa mais que polimento.

Nessas zonas, um prompt claro somado a uma revisão cuidadosa vence a digitação manual sempre.

O fio condutor é que essas tarefas são **bem conhecidas e verificáveis**. O modelo já viu dez mil endpoints CRUD, então produz versões idiomáticas; e você consegue dizer de relance se o resultado está certo. Quando essas duas coisas são verdadeiras — padrão comum, barato de checar — o vibe coding é quase puro ganho.

Onde falha

Seja honesto sobre os limites, porque é aqui que o vibe coding sem supervisão queima as pessoas:

- **Lógica profunda e inédita.** Algoritmos sutis, concorrência, qualquer coisa onde estar 95% certo significa estar quebrado. O modelo é confiante mesmo quando está errado.
- **Mudanças espalhadas por muitos arquivos.** Ele perde o fio da meada em uma base de código grande e quebra coisas silenciosamente que não consegue enxergar.
- **Segurança e dinheiro.** Autenticação, pagamentos, permissões, qualquer coisa onde um bug com aparência plausível tem consequências reais. Revise isso como se seu emprego dependesse — porque pode ser que dependa.
- **Problemas subespecificados.** Se você não sabe como é o “correto”, a IA não pode ler sua mente. Especificação ruim entra, código ruim sai.

O modo de falha é quase sempre o mesmo: código que *parece* certo, roda no caminho feliz, e está errado de um jeito que você teria pego se tivesse lido. A correção é o passo de revisão. Não há como pular isso.

Duas perguntas posicionam qualquer tarefa neste mapa — o padrão é comum, e o resultado é barato de checar:

	EASY TO VERIFY	HARD TO VERIFY
COMMON	PURE UPSIDE	go slower,

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

Repare no inverso da regra acima: esses são os casos que são *raros, sutis ou caros de verificar*. O padrão é incomum, então o modelo está chutando; ou o bug é invisível no caminho feliz, então uma checagem rápida não vai pegá-lo. Isso não significa evitar a IA aqui — significa desacelerar, encolher os passos, exigir testes e ler cada linha como se ela fosse hostil.

Armadilhas comuns

Algumas armadilhas pegam quase todo mundo no primeiro mês:

- **Aceitar a primeira coisa que roda.** “Funciona” e “está correto” são afirmações diferentes. A primeira significa que o caminho feliz não travou; a segunda significa que você checou as bordas. Não confunda as duas.
- **Mega-prompts.** Pedir uma funcionalidade inteira de uma vez dá ao modelo espaço demais para se desviar e dá a você coisa demais para revisar de uma só vez. Decomponha. Publique a menor fatia revisável, e depois construa em cima dela.
- **Deixar o contexto apodrecer.** À medida que uma sessão cresce, o modelo começa a contradizer decisões anteriores e reintroduzir bugs que você já tinha corrigido. Quando isso começar, resuma o estado atual e recomece do zero em vez de brigar com uma conversa confusa.
- **Confiar na confiança como sinal.** O tom do modelo é idêntico esteja ele certo ou alucinando. A certeza dele não te diz nada. Sua verificação te diz tudo.
- **Terceirizar a decisão, não só a digitação.** Tudo bem perguntar “quais são minhas opções para X?”. É um erro deixar a resposta escolher sua arquitetura sem o seu julgamento no circuito.

Bom senso é a nova habilidade central

Quando qualquer um pode gerar código, a habilidade escassa é saber qual código gerado é *bom*.

Bom senso é o que te diz:

- Essa abstração é prematura; peça a versão mais simples.
- Isso resolve a demo, mas não a entrada real.
- Isso está tecnicamente correto, mas é um pesadelo de manutenção.
- Isso está *quase* certo — e aqui está a coisa específica a mudar.

Você constrói bom senso da maneira tradicional: lendo muito código, publicando coisas e vendo o que quebra. A IA não substitui essa experiência — ela a torna mais valiosa, porque agora o seu julgamento é o gargalo e o diferencial. A pessoa com bom senso publica software sólido rápido. A pessoa sem ele publica rápido e depois se afoga em bugs.

Aqui está o corolário desconfortável: a IA eleva o piso para todo mundo, o que significa que ela *achata* a vantagem de saber sintaxe e *afia* a vantagem de ter julgamento. Os juniores que se apoiam nela para pular o aprendizado vão estagnar, porque nunca constroem o bom senso que diz a eles quando o resultado está errado. Os que a usam para publicar mais e ler mais vão compor ganhos. Mesma ferramenta, resultados opostos — e a diferença é inteiramente se você permanece engajado com o que volta.

Expectativas realistas

Defina isso agora para não se decepcionar depois:

- **Você vai ler muito código.** Revisar é o trabalho. Se você não quer ler código, isto não é um atalho para escapar disso.
- **Não vai acertar de primeira nada não trivial.** Espere algumas rodadas do ciclo. Isso é normal, não fracasso.
- **Você ainda é responsável.** Quando código escrito por IA quebra em produção, “o modelo escreveu” não é uma defesa. Você publicou.
- **Fica mais rápido, não grátis.** As ferramentas são notáveis. Não são mágica. O julgamento continua sendo seu.

E para ser direto sobre o que este livro *não* é: não é um caminho para publicar software que você não entende, e não é um esquema de enriquecimento rápido. Quem vende vibe coding como “construa um app sem aprender a programar” está vendendo a versão brinquedo — aquela que desmorona no contato com usuários reais. A habilidade se compõe para as pessoas que a tratam como alavanca sobre o julgamento de engenharia, não como substituta dele.

A promessa do vibe coding é real: você pode construir mais, mais rápido, em mais domínios do que conseguiria sozinho. Mas ela recompensa o engenheiro que permanece engajado — que direciona, revisa e refina — e pune aquele que se desliga e aceita o que quer que apareça.

Seja o tech lead, não o espectador. O resto deste livro é sobre como.

Recap and Practice

Principais conclusões

- Vibe coding é direcionar uma IA para construir software enquanto você continua no comando do *o quê* e do *porquê* — não publicar cegamente código que você não consegue explicar.
- A regra que mantém tudo coeso é *entenda o que você publica*: você não precisa ter digitado, mas precisa ser capaz de defendê-lo e consertá-lo.
- Seu gargalo se moveu da produção (digitar rápido) para a especificação e a revisão (descrever com clareza, julgar com crítica).
- Viva no ciclo — intenção → gerar → revisar → refinar — e nunca pule o passo de revisão, onde a engenharia de fato acontece.
- A confiança do modelo não é um sinal; sua verificação é. Bom senso e julgamento são as habilidades escassas que se compõem.

Experimente

Escolha uma função pequena que você genuinamente entenda — algo como “validar se uma senha atende a uma política” ou “formatar uma quantidade de bytes como um texto legível”. Escreva um prompt preciso para ela (assinatura, regras, casos de borda, um pedido de testes), gere-a, e então deliberadamente execute o passo de *revisão*: percorra pelo menos um caso de borda manualmente e confirme que o código o trata. Se não tratar, escreva um prompt de refinamento preciso (sintoma + causa + correção) em vez de “conserta aí”. O objetivo é sentir o ciclo completo uma vez, do início ao fim, em um código que você consegue avaliar por completo.

Prompt do 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.

Configurando Seu Ambiente de Trabalho com IA



Vibe coding é uma parceria. Você traz a intenção e o julgamento; a IA traz velocidade e digitação incansável. Mas, como qualquer parceria, ela funciona melhor com um ambiente compartilhado, regras claras e um lugar seguro para errar. Este capítulo trata de construir esse ambiente antes de você escrever uma única funcionalidade.

Um bom ambiente de trabalho faz três coisas: dá à IA o contexto necessário para fazer mudanças inteligentes, permite que você veja os resultados dessas mudanças rapidamente e torna toda mudança fácil de desfazer. Acerte nisso e a IA parece uma colaboradora sênior. Erre e ela parece uma máquina caça-níqueis. A diferença raramente está no modelo — está na sala em que você o coloca. Um agente competente em um repositório caótico, sem testes, sem arquivo de contexto e sem controle de versão, vai se atrapalhar; um agente modesto em um projeto limpo e bem instrumentado vai te surpreender. A configuração é o multiplicador.

Nada disso é trabalho burocrático que você faz uma vez e esquece. Trate seu ambiente de trabalho como algo que você ajusta continuamente: toda vez que a IA te surpreender de forma ruim, pergunte-se se faltou

um pedaço de contexto, se o ciclo de feedback estava lento, ou se uma configuração padrão insegura permitiu que isso acontecesse — e então conserte a sala, não apenas a linha de código.

Escolha um Editor ou Assistente Nativo de IA

Sua primeira decisão é onde a IA vai morar. Hoje existem, grosso modo, três categorias, e você não precisa se comprometer para sempre.

- **Editores nativos de IA** (Cursor, Windsurf): um editor de código familiar com a IA profundamente integrada. Ideal se você quer ver e mexer no código enquanto a IA trabalha ao seu lado.
- **Agentes de terminal/CLI** (Claude Code, Codex CLI, Gemini CLI, Aider): pense em digitar comandos em um prompt em vez de clicar em botões — esse painel de controle só-texto é a linha de comando, ou CLI (Command-Line Interface, Interface de Linha de Comando). A IA roda como um agente de linha de comando que lê arquivos, os edita e executa comandos. Ideal para deixar a IA trabalhar de forma mais autônoma em muitos arquivos.
- **Plugins de IDE** (GitHub Copilot, Continue, JetBrains AI): uma IDE (Integrated Development Environment, Ambiente de Desenvolvimento Integrado) é o aplicativo tudo-em-um em que a maioria dos programadores escreve código — esses plugins são assistência acoplada dentro de um editor que você já ama.

Para lançar software de verdade, escolha uma ferramenta que consiga **ler todo o seu projeto, editar múltiplos arquivos e executar comandos** (testes, builds, linters) por conta própria. A capacidade de rodar e conferir o próprio trabalho é o que separa um assistente útil de um brinquedo de autocompletar. Uma ferramenta que só sugere a próxima linha deixa toda a verificação na sua mão; uma ferramenta que roda o teste que acabou de alterar fecha o ciclo sozinha.

Além dessa capacidade central, vale a pena avaliar alguns outros pontos antes de se estabelecer:

- **Janela de contexto e percepção do projeto** — a ferramenta consegue de fato carregar os arquivos relevantes, ou esquece o início da conversa na metade de uma tarefa?
- **Modelo de permissões** — ela pergunta antes de executar comandos e editar arquivos, ou age primeiro? Ambos têm seu lugar; saiba em qual modo você está.
- **Custo e limites** — agentes autônomos podem consumir uso rapidamente. Entenda o modelo de preços antes de deixar um rodando por uma hora.

Não se martirize com a escolha; as habilidades deste livro transferem-se entre ferramentas. A maioria dos vibe coders experientes acaba mantendo duas instaladas — um agente de terminal rápido para trabalho em volume e um editor gráfico para revisão. Uma GUI (Graphical User Interface, Interface Gráfica do Usuário) é o tipo familiar de programa de apontar-e-clicar, com janelas e botões; uma TUI (Text User Interface, Interface de Usuário em Texto) é sua prima só-texto que vive no terminal. A seção final do capítulo, sobre TUI vs. GUI, retoma exatamente essa dupla.

Estruture Seu Projeto Para Que a IA Consiga Navegar Nele

Modelos de IA raciocinam sobre seu código lendo-o. Um projeto bagunçado, disperso e inconsistente é tão difícil para eles quanto é para você. Você não precisa de uma arquitetura perfeita, mas alguns hábitos compensam imediatamente:

- **Mantenha um layout previsível.** Código-fonte em um só lugar, testes ao lado ou espelhando o código-fonte, configuração na raiz.
- **Use nomes claros e descritivos.** `calculateMonthlyInvoiceTotal` é melhor que `calc2`. A IA usa nomes como pistas.
- **Prefira arquivos menores.** Um arquivo de 200 linhas é mais fácil de alterar corretamente do que um de 2.000, tanto para você quanto para ela.
- **Coloque coisas relacionadas juntas.** Quando o código, os estilos e os testes de uma funcionalidade ficam juntos, a IA reúne contexto em uma única leitura.

- **Mantenha um ponto de entrada claro.** Um único lugar óbvio onde o aplicativo começa dá à IA um fio para puxar quando estiver mapeando o projeto.

Um layout típico e amigável para IA pode se parecer com isto:

```
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
```

O ganho é cumulativo: quanto mais fácil for para a IA navegar pelo seu projeto, menores e mais cirúrgicas se tornam suas mudanças. Um repositório bem organizado permite que a IA mexa em três arquivos relevantes em vez de tentar adivinhar entre trinta — o que significa diffs menores, menos acidentes e revisões que você realmente consegue ler. Se você perceber a IA fazendo edições amplas e sem foco, muitas vezes o verdadeiro culpado é a estrutura, não o prompt.

Escreva um Arquivo de Contexto (AGENTS.md)

A coisa de maior alavancagem que você pode fazer é escrever um **arquivo de contexto** que a IA lê automaticamente. Ferramentas diferentes procuram nomes diferentes — AGENTS.md, CLAUDE.md, .cursor-rules, ou um .github/copilot-instructions.md — mas a ideia é idêntica: um documento vivo que conta à IA como o seu projeto funciona e como você quer que ela se comporte.

Sem ele, a IA chuta. Com ele, a IA para de reformatar seu código, para de inventar bibliotecas que você não usa e para de reexplicar decisões que você já tomou. Pense nisso como documentação de integração para um novo colega de equipe brilhante, mas com amnésia total entre sessões — tudo que ele precisa para ser produtivo tem que estar em algum lugar que ele vai ler toda vez.

Um arquivo inicial prático:

```
# 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.
```

Mantenha-o curto e atualizado. Um arquivo de contexto que cresce até mil linhas deixa de ser lido com atenção — por você ou pelo modelo — então pode-o com a mesma agressividade com que você adiciona coisas a ele. Quando a IA repetidamente fizer a mesma suposição errada, esse é o seu sinal para adicionar uma linha; quando uma regra não corresponder mais à realidade, apague-a antes que comece a enganar ativamente o agente.

Alguns padrões separam um arquivo de contexto que funciona de um que é ignorado:

- **Seja específico e concreto.** “Use nossas convenções” não serve para nada. “Use zod para validação; não usamos yup” é uma regra que a IA de fato consegue seguir.
- **Diga os “não faça” com tanta força quanto os “faça”.** Modelos são ansiosos para ajudar, o que significa que vão adicionar uma dependência ou reescrever uma configuração de bom grado, a menos que você diga para não fazerem isso.
- **Aponte para exemplos.** “Siga o estilo de src/features/invoices” dá à IA um modelo concreto para copiar, em vez de uma instrução vaga para interpretar.

Trate o arquivo de contexto como um registro de toda lição que, sem ele, você teria que repetir. É o investimento mais barato e mais duradouro deste capítulo inteiro.

Faça do Controle de Versão um Reflexo

Controle de versão é sua rede de segurança, e com IA ele se torna essencial. A IA pode alterar dez arquivos em cinco segundos — você precisa de uma forma limpa de inspecionar, manter ou descartar esse trabalho.

- **Faça commits com frequência, em pedaços pequenos.** Antes de pedir uma mudança grande, faça commit do que está funcionando. Uma árvore de trabalho limpa significa que você sempre pode voltar a ela.
- **Leia o diff antes de fazer commit.** Essa é sua verdadeira etapa de revisão. Você não precisa entender cada linha, mas deveria entender o que mudou e por quê.
- **Use branches para trabalho arriscado.** Crie um branch, deixe a IA soltar a franga, e faça merge só se gostar do resultado. Se não, apague-o — sem prejuízo.
- **Escreva mensagens de commit honestas.** Tanto o seu eu futuro quanto a IA leem o histórico para entender a intenção.

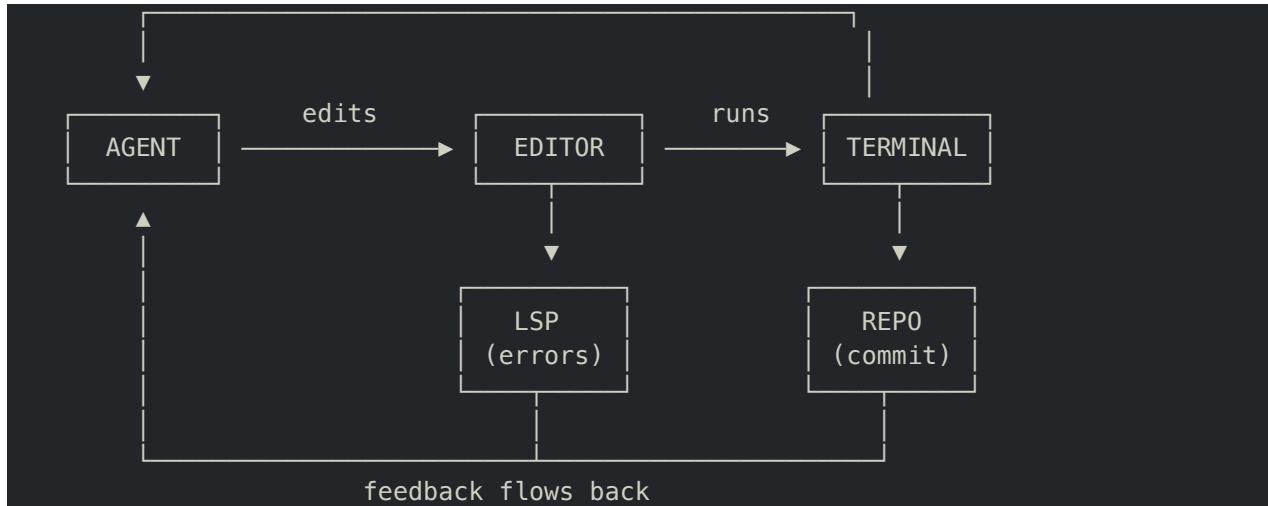
A mudança de mentalidade: commits não são burocracia, são pontos de salvamento em um jogo. Salve antes da batalha contra o chefe. E aprenda as duas saídas de emergência que fazem um histórico limpo valer a pena manter — `git checkout .` (ou `git restore .`) para descartar experimentos não commitados, e `git revert` para desfazer um commit que você já fez. Saber que você sempre pode voltar a um estado conhecido e bom é exatamente o que permite dar à IA uma corda longa sem ansiedade.

Mantenha o Ciclo de Feedback Curto

Quanto mais rápido você consegue ver se uma mudança funciona, mais rápido você e a IA convergem para algo correto. Um ciclo lento — em que você espera minutos para saber se algo quebrou — multiplica silenciosamente todo erro.

- **Faça de “rodar o projeto” um único comando.** `npm run dev`, `make test`, seja lá o que for, documente no seu arquivo de contexto para que a IA também consiga rodar.
- **Aposte em testes rápidos.** Mesmo um punhado de testes unitários dá à IA uma forma de verificar o próprio trabalho sem você precisar ficar de babá.
- **Use um linter e um formatador.** Eles pegam toda uma classe de erros instantaneamente e mantêm discussões de estilo fora das suas revisões.
- **Mostre os erros de volta para a IA.** Quando algo quebrar, cole a mensagem de erro real. Feedback específico produz correções específicas; “não funcionou” produz chutes.

As peças de um ambiente de trabalho com IA formam um único ciclo conectado — o agente edita o repositório, o editor e o language server reportam problemas, o terminal executa o código, e todo resultado flui de volta para o agente:



A versão mais profunda dessa ideia é deixar a IA fechar o ciclo sozinha. Em vez de você rodar o comando, ler a falha e repassá-la, dê ao agente permissão para rodar os testes e o verificador de tipos após cada mudança e reagir ao que ele vê. O ciclo se torna: editar, rodar, ler a falha, corrigir, rodar de novo — tudo sem você no meio. Seu trabalho muda de mensageiro para diretor: você define a meta e as barreiras, e depois observa o ciclo girar. Quando a IA consegue editar, rodar, ver a falha e corrigi-la sozinha, você construiu um ciclo que faz trabalho de verdade enquanto você permanece na cadeira de diretor.

Gerencie Segredos com Cuidado

Ferramentas de IA leem seus arquivos e, às vezes, compartilham contexto com um serviço remoto. Isso torna o vazamento de segredos um risco real. Construa bons hábitos desde o primeiro dia:

- **Nunca coloque chaves reais no código.** Use variáveis de ambiente e um arquivo `.env`.
- **Sempre coloque seu `.env` no gitignore.** Faça commit de um `.env.example` com placeholders vazios, para que a IA saiba o que é necessário sem ver os valores.
- **Delimite as chaves com escopo estreito** e rotacione qualquer coisa que possa ter sido exposta.
- **Diga à IA, no seu arquivo de contexto,** para nunca fixar segredos no código nem fazer commit deles.

Uma regra bem enfática como `Never read or print the contents of .env` no seu `AGENTS.md` é um seguro barato. Assim como um hook de pre-commit ou uma ferramenta de varredura de segredos que recusa um commit contendo algo parecido com uma chave — defesa em profundidade importa aqui porque o custo de um vazamento é assimétrico. Uma chave enviada a um repositório público está comprometida no exato momento em que chega lá, mesmo que você a apague segundos depois, porque robôs vasculham commits em tempo real. Na dúvida, presuma que uma chave exposta está queimada e rotacione-a; rotacionar uma chave leva um minuto, enquanto limpar a bagunça depois de um vazamento

pode levar dias.

Uma Configuração Local Sensata Torna as Mudanças da IA Seguras

Amarre tudo isso com um ambiente onde erros são baratos e recuperáveis:

- Trabalhe em um **branch**, não diretamente na `main`.
- Mantenha a árvore de trabalho **limpa antes** de mudanças grandes, para que o diff tenha sentido.
- Tenha os testes e um servidor de desenvolvimento **prontos para rodar** com um único comando.
- Mantenha os segredos **fora do repositório** e o arquivo de contexto **atualizado**.

Com essa base, o pior cenário para qualquer mudança da IA é um `git checkout` e um recomeço. Essa segurança é o que permite que você se mova rápido e dirija com ousadia — sabendo que nada do que a IA faz é verdadeiramente irreversível. No próximo capítulo, vamos começar a usar esse ambiente de trabalho para de fato construir.

Diagnósticos ao Vivo (LSP)

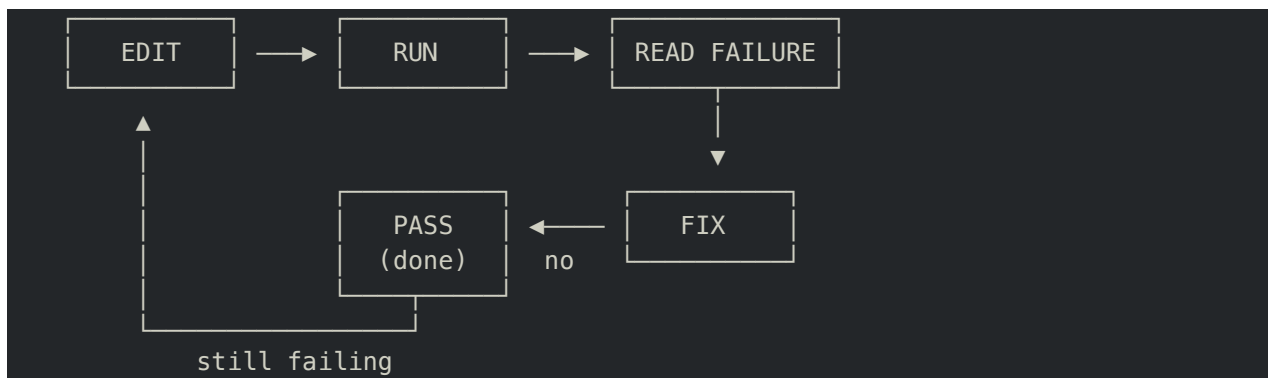
O ciclo de feedback mais curto não espera você rodar o código — ele pega os erros no exato momento em que são digitados. É isso que o **Language Server Protocol (LSP)** fornece: a “inteligência” viva por trás do seu editor, incluindo **erros, informações de tipo, ir-para-definição e autocompletar** em tempo real. Um language server (servidor de linguagem) lê seu código enquanto ele muda e reporta problemas antes que qualquer coisa seja executada.

A alavancagem para o vibe coding é **realimentar esses diagnósticos para a IA**. Sem isso, o ciclo é lento: a IA escreve código, você o executa, ele trava por causa de um erro de digitação ou de tipo, e você cola a mensagem de volta manualmente. Com diagnósticos conectados, o agente vê o mesmo sublinhado vermelho que você vê no instante em que escreve a linha — e corrige o erro de tipo ou de lint *no ato*, antes mesmo de o código rodar.

Você pode dar essa visão ao agente de algumas formas:

- Rode o language server do seu stack (tsserver, rust-analyzer, pyright, e assim por diante) — a maioria dos editores nativos de IA faz isso por você.
- Faça o agente rodar um comando de verificação de tipos ou de lint e ler a saída, ou exponha os diagnósticos por meio de um servidor MCP.
- Trate um relatório de diagnósticos limpo como parte do “concluído”, não como um adendo.

Conectado dessa forma, o agente roda o mesmo ciclo autocorretivo por conta própria — editar, rodar, ler a falha, corrigir e rodar de novo até que os diagnósticos voltem limpos:



É por isso também que uma linguagem tipada pode ser um superpoder discreto para o vibe coding: cada anotação de tipo é uma pequena especificação verificada por máquina que pega os erros da IA antes de eles chegarem a rodar. Um ciclo de rodar-travar-colar se torna um ciclo silencioso e autocorretivo. A IA gira menos em falso e converge mais rápido.

Estendendo Seu Agente

Fora da caixa, um assistente de IA é genérico. O mesmo modelo base se torna uma colaboradora muito melhor assim que você envolve ferramentas ao redor dele — transformando um chatbot simples em um **harness** (arreio, estrutura de restrições) ajustado ao seu projeto. Os principais pontos de extensão a conhecer:

- **Skills** — capacidades empacotadas e reutilizáveis (instruções mais scripts) que o agente carrega sob demanda, de modo que o conhecimento específico do seu domínio fica disponível sem precisar ser reexplicado a cada sessão.
- **Plugins** — pacotes que empacotam skills, comandos, hooks e conectores juntos como uma única unidade instalável.
- **MCP (Model Context Protocol)** — um padrão aberto para conectar o agente a ferramentas e dados externos (bancos de dados, APIs (Application Programming Interfaces, Interfaces de Programação de Aplicações — os menus padronizados de comandos que outros softwares expõem), seu sistema de arquivos) através de uma interface comum única.
- **Subagentes** — agentes auxiliares especializados que o agente principal cria para trabalho focado (busca, revisão, uma missão paralela longa), cada um com sua própria janela de contexto.
- **Hooks** — scripts que disparam automaticamente em eventos do ciclo de vida (antes de um comando rodar, depois de um arquivo ser editado), permitindo que você imponha regras ou injete contexto de forma determinística.
- **Comandos de barra (slash commands)** — fluxos de trabalho curtos e nomeados que você invoca com /nome, condensando um pedido repetido de múltiplas etapas em uma única tecla.

Você não precisa de tudo isso no primeiro dia. Comece simples e adicione uma skill ou um comando de barra no momento em que perceber que está repetindo a mesma configuração. Uma regra prática útil: na terceira vez que você digitar a mesma instrução de várias etapas, transforme-a em um comando de barra; na primeira vez que você desejar que o agente tivesse acesso a uma ferramenta que não tem, procure um servidor MCP para isso. Hooks, em particular, retornam no capítulo sobre barreiras de proteção como uma forma de *impor* suas regras em vez de apenas esperar que a IA se lembre delas.

macOS vs. Windows

A maior parte das ferramentas de IA — agentes de CLI, scripts, tutoriais — silenciosamente presume um shell Unix. Isso molda a escolha prática de ambiente de desenvolvimento.

- O **macOS é nativo em Unix**. Ele vem com um shell POSIX de verdade e os mesmos `git`, `ssh` e `curl` que o resto do mundo dev usa. A maioria dos agentes e scripts “simplesmente funciona”.
- O **Windows** é totalmente capaz, mas o caminho mais suave é o **WSL2** (Windows Subsystem for Linux, Subsistema Windows para Linux): um ambiente Linux de verdade dentro do Windows. Faça sua programação com IA *dentro* do WSL2, em vez de lutar contra as peculiaridades do PowerShell nativo.
- O **Linux** é o terceiro membro da família e se comporta quase exatamente como o macOS para esses fins — ele é Unix, então os mesmos agentes e scripts rodam sem alteração. Se você já está nele, está pronto.

Algumas armadilhas para manter seu trabalho portátil entre todos eles:

- **Caminhos de arquivo** — `/home/voce/projeto` versus `C:\Users\voce\projeto`; barras invertidas e letras de unidade quebram scripts. O WSL2 evita a maior parte disso.
- **Terminadores de linha** — o Windows usa CRLF, o Unix usa LF. Incompatibilidades causam diffs ruidosos e scripts de shell quebrados; configure `core.autocrlf` e um `.gitattributes` para normalizar para LF.
- **Sensibilidade a maiúsculas/minúsculas** — macOS e Windows frequentemente tratam `File.ts` e `file.ts` como o mesmo arquivo; Linux e CI não, então código que compila localmente pode falhar na nuvem. Seja consistente com maiúsculas e minúsculas em imports e nomes de arquivo.

- **Permissões** — os modos de arquivo do Unix (`chmod +x`) não mapeiam de forma limpa para o Windows.

Seja qual for o seu sistema, peça à IA para manter os comandos portáteis — barras normais, terminadores de linha LF, capitalização consistente — para que seus scripts rodem igual no seu laptop e no CI. Quando a IA te entregar um one-liner que presume uma ferramenta que você não tem, o jeito mais rápido de resolver costuma ser dizer a ela o seu sistema operacional e shell no arquivo de contexto, para que ela pare de chutar.

TUI vs. GUI

Ferramentas de codificação com IA vêm em dois formatos, e um vibe coder produtivo recorre a ambos. **Agentes de terminal/TUI** (Claude Code, Codex CLI, Aider) rodam no terminal: rápidos, roteirizáveis (scriptable) e combináveis, ótimos para automação e para deixar a IA trabalhar de forma autônoma em muitos arquivos. **Ferramentas de GUI/editor** (Cursor, Windsurf, extensões de IDE) vivem em um editor visual: descobríveis, com diffs inline, aceitar-com-um-clique e uma curva de aprendizado suave — mais fáceis para revisar mudanças.

	TUI / terminal	GUI / editor
Velocidade	Muito rápido, baixa sobrecarga	Mais lento, mais interface
Roteirizável (scriptable)	Sim — encadear, automatizar	Limitado
Diffs visuais	Texto simples	Ricos, inline
Descobribilidade	Baixa (é preciso conhecer os comandos)	Alta (clicável)
Melhor para	Automação, autonomia, usuários avançados	Revisão, integração, trabalho visual

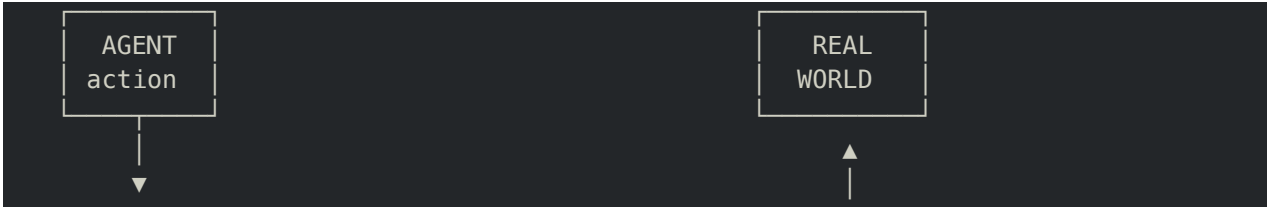
Você não precisa escolher um para sempre. A combinação é o ponto central: conduza o grosso do trabalho por meio de um agente de TUI rápido, depois entre em uma GUI para revisão visual ou edições de granularidade fina. Um fluxo de trabalho comum é deixar um agente de terminal fazer as mudanças amplas em muitos arquivos, e depois abrir o diff lado a lado da GUI para de fato ler e aprovar o que foi feito — velocidade para fazer, clareza para conferir. Escolha uma ferramenta principal com a qual você se sinta confortável, e lembre-se de que as habilidades deste livro se transferem entre elas.

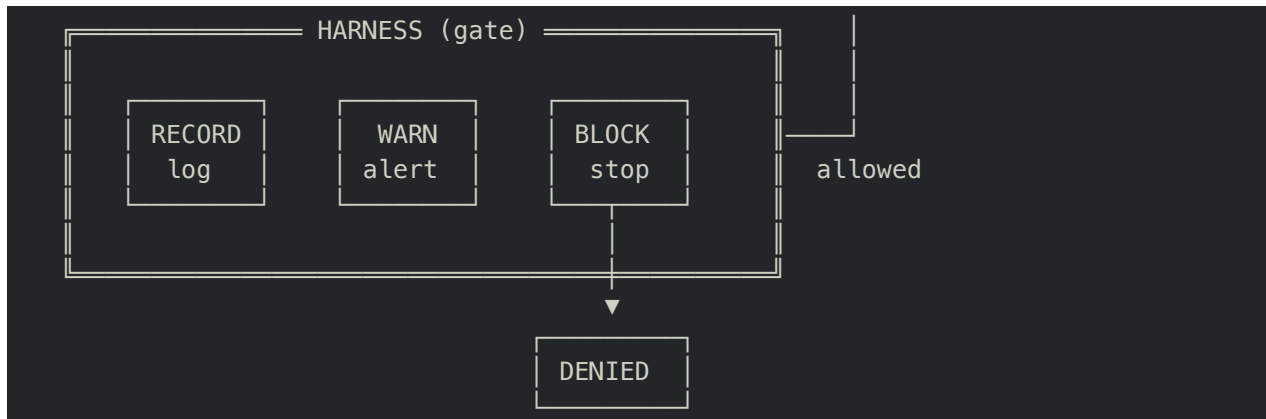
Engenharia de Harness

Hábitos são coisas que você precisa se lembrar de fazer. O próximo degrau na configuração do seu ambiente de trabalho é transformar essas verificações em algo que a IA **não pode** pular. Isso é um harness.

Pense em um harness como um posto de controle de segurança que fica entre o agente e o mundo real. A cada etapa perigosa, ele faz uma de três coisas: registra o que aconteceu, avisa você, ou bloqueia a ação por completo. Antes de um comando rodar, antes de um arquivo ser escrito, quando um prompt é enviado, no commit, no push — sempre há um portão. O agente não consegue contorná-lo, do jeito que você poderia esquecer de rodar os testes às duas da manhã.

Toda ação perigosa que o agente realiza passa primeiro pelo harness, onde cada portão pode registrá-la, avisar você ou bloqueá-la antes que ela chegue ao mundo real:





Por que tornar isso obrigatório? A aposta inteira do vibe coding é “deixe a IA correr, você dirige”. Isso só compensa se a IA não conseguir sair do caminho e cair de um penhasco enquanto você pisca. Um harness move sua segurança da *vigilância* (ler cada diff, lembrar cada regra) para o *código* (regras aplicadas automaticamente). Para qualquer coisa que você colocaria na frente de um usuário real, isso não é um capricho opcional — é o que torna a autonomia segura.

Um bom harness segue alguns princípios que vale a pena conhecer:

- **Silencioso no sucesso, barulhento na falha** — ele não diz nada quando tudo está bem, então um aviso ainda significa alguma coisa.
- **Nunca corrigir automaticamente** — ele propõe, bloqueia ou avisa; não “corrige” silenciosamente o agente escondendo o problema.
- **Aposente regras dormentes antes de adicionar novas** — mantenha-o enxuto e confiável, não uma pilha crescente de verificações obsoletas.
- **Guiado por configuração** — um único motor, configurações por projeto, para que se encaixe em qualquer repositório.
- **Logs somente de anexação (append-only)** — um registro à prova de adulteração do que o agente realmente fez.

Você não precisa construir isso do zero. Existem harnesses prontos, plug-and-play — dancin-lab/harness é um motor agnóstico de projeto que você pode conectar ao ciclo de vida de ferramentas de um agente de codificação. Sinceramente: um harness não torna a IA correta. Ele torna os erros da IA baratos de detectar — o que, quando você está se movendo rápido, já é a maior parte da batalha.

Segurança de Custos: CLI vs. API

Uma única fatura pode arruinar um projeto, e raramente é a hospedagem a culpada. É uma API de IA cobrada por uso. As ferramentas que você configura para programar no dia a dia — Claude Code, Cursor e afins — normalmente são uma **assinatura de taxa fixa**: o custo de mais um prompt é efetivamente zero. No momento em que o seu *aplicativo* chama uma **API ou SDK** de IA bruta — sendo um SDK (Software Development Kit, Kit de Desenvolvimento de Software) a biblioteca de código pré-empacotada que um provedor te dá para chamar o serviço dele — a economia se inverte: você é cobrado **por token, por requisição**, sem teto. Um retry com bug, um job em lote sem limite, ou um agente que entra em loop podem silenciosamente transformar uma fatura de \$0 em ociosidade em uma surpresa de quatro dígitos da noite para o dia. Isso é o “susto da fatura” (“bill shock”), e é a forma mais comum de builders solo se queimarem.

Coloque uma jaula em volta de qualquer API paga antes de lançar:

- **Defina um teto de gastos rígido** no painel do provedor — um teto de verdade, não apenas um alerta por e-mail.

- **Prefira a CLI de taxa fixa para o desenvolvimento;** reserve a API cobrada por uso para a funcionalidade de produção que realmente precisa dela.
- **Limite a taxa das suas próprias chamadas** — capeie a concorrência e as requisições por minuto — e observe o painel de uso nos primeiros dias.
- **Use modelos baratos/pequenos para trabalho em volume, e teste com entradas minúsculas primeiro** (um registro, não a tabela inteira).
- **Nunca lance um loop sem estrangulamento que chama uma API paga.** Limite todo loop.

Faça disso uma regra desde o início: *toda API paga tem um teto rígido e um estrangulamento (throttle).*

Recapitulação e Prática

Principais conclusões

- Um bom ambiente de trabalho dá à IA contexto, um ciclo de feedback rápido e a possibilidade de desfazer barato — a configuração é o multiplicador, muitas vezes mais do que o próprio modelo.
- Um arquivo de contexto (AGENTS.md / CLAUDE.md) é a coisa de maior alavancagem que você pode escrever: regras concretas, “não faça” enfáticos e ponteiros para exemplos.
- Faça do controle de versão um reflexo — commits pequenos, leia o diff, use branches para trabalho arriscado — para que nada do que a IA faz seja verdadeiramente irreversível.
- Mantenha o ciclo curto e deixe o agente fechá-lo: editar, rodar, ler a falha, corrigir. Realmente diagnósticos (LSP, lint, verificação de tipos) de volta para a IA.
- Coloque uma jaula em toda API paga com um teto rígido e um estrangulamento, mantenha segredos fora do repositório e use um harness para transformar verificações de segurança em algo que a IA não possa pular.

Experimente

Abra qualquer projeto em que você esteja trabalhando (ou monte um vazio) e escreva à mão um primeiro rascunho de AGENTS.md: o que é o projeto, o stack de tecnologia, os comandos exatos para rodar dev/test/lint, três convenções reais e pelo menos dois “não faça” enfáticos (por exemplo, “nunca leia ou imprima o conteúdo de .env”, “não adicione dependências sem avisar antes”). Depois peça à sua ferramenta de IA para fazer uma mudança trivial e observe se ela respeita o arquivo. Cada vez que ela fizer algo que você não queria, adicione ou refine uma linha. Você está ajustando a sala, não apenas o código.

Prompt do capítulo

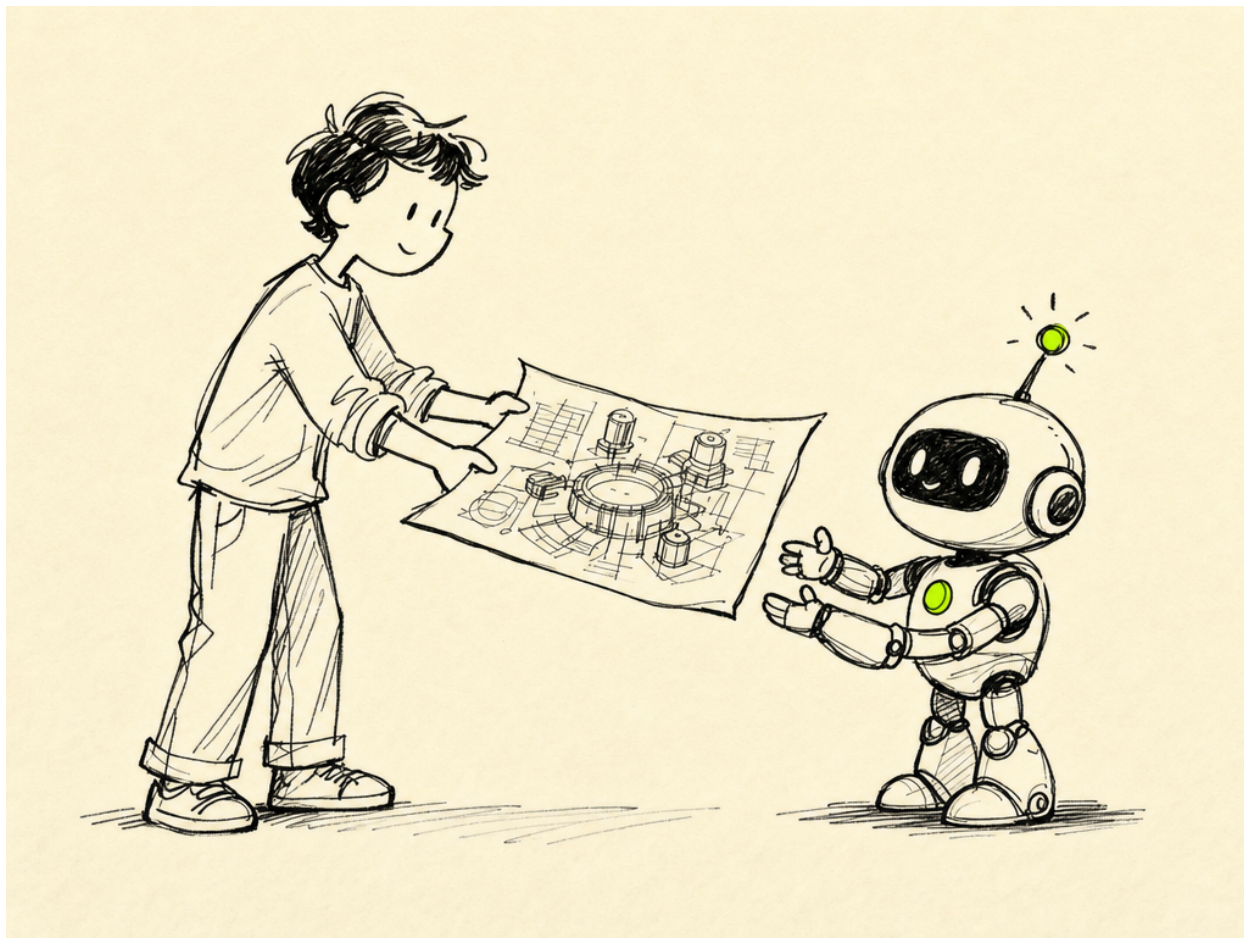
```
Read my project and help me write a concise AGENTS.md context file for it.
```

```
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.
```

Prompt Como um Engenheiro



A diferença entre uma sessão frustrante com IA e uma produtiva raramente está no modelo. Está no prompt. A maioria dos iniciantes trata a IA como um mecanismo de busca — digita um desejo vago e espera pelo melhor. Engenheiros experientes a tratam como uma colega júnior afiada, mas literal: alguém rápida e capaz, mas que vai construir exatamente o que você descrever, incluindo os seus erros.

Este capítulo trata de fechar essa lacuna. Você não precisa memorizar truques ou colecionar uma pasta de templates de prompt “mágicos”. Precisa se comunicar do jeito que um bom engenheiro comunica uma tarefa: com contexto, restrições e uma definição clara de “pronto”. Tudo abaixo é uma variação dessa única habilidade.

Dê Contexto Antes de Dar a Tarefa

A IA não consegue ver seu projeto como você vê. Ela não sabe a versão do seu framework, a estrutura dos seus arquivos, nem as convenções que você já estabeleceu. Quando você pula essa etapa, ela adivinha — e adivinha mal. Pior: ela adivinha com *confiança*, então a saída parece plausível até colidir com o seu código real.

Compare estes dois prompts para o mesmo pedido.

```
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.
```

O primeiro prompt produz um código genérico que pode usar uma regex que você não quer, em um estilo que não bate com o seu código. O segundo produz um código que você pode simplesmente colar. A regra: **declare a stack, as convenções e as peças existentes que a IA deve reutilizar.**

Um bom prompt empilha essas peças em ordem, de baixo para cima — contexto na base, o pedido em si perto do topo:

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 ▲

Uma checklist mental útil antes de apertar enter: linguagem e framework (com versão, se importar), os arquivos ou módulos relevantes, os padrões já existentes no projeto, e quaisquer bibliotecas que a IA deveria preferir ou evitar. Você não precisa dos quatro itens toda vez — mas se a resposta mudasse dependendo de um deles, nomeie-o. Na dúvida, cole a assinatura, o tipo ou a configuração real com a qual o novo código precisa se encaixar. Algumas linhas de código existente ensinam mais sobre suas convenções à IA do que um parágrafo inteiro de descrição.

Especifique Entradas, Saídas e Casos de Borda

Uma tarefa só está definida quando seus limites estão definidos. Engenheiros experientes pensam em termos de “o que entra, o que sai, e o que acontece quando algo dá errado”. Incorpore isso ao 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.
```

Quando você nomeia os casos de borda, a IA os trata. Quando você não nomeia, ela escreve o caminho feliz e você descobre as lacunas em produção. A mesma disciplina se aplica além de funções puras: para um endpoint, especifique a resposta de sucesso e as respostas de erro (qual código de status em caso

de e-mail duplicado? em caso de corpo malformado?). Para um componente de interface, especifique o estado vazio e o estado de carregamento, não só o estado populado. O comportamento interessante da maioria dos softwares vive nas bordas, então é ali que seu prompt deve gastar suas palavras.

Se você não tiver certeza de quais são os casos de borda, pergunte primeiro à IA: “*What edge cases should a date parser handle?*” — e depois devolva a resposta como restrições. Isso transforma o modelo de um gerador de código em um gerador de checklist, e revisar uma checklist é mais rápido do que depurar os casos que você esqueceu.

Trabalhe em Passos Pequenos

O erro mais comum é pedir demais de uma vez. “Build me a user dashboard with auth, charts, and a settings page” vai produzir uma parede de código que você não consegue revisar, com bugs espalhados por arquivos que você nem leu.

Passos pequenos mantêm você no controle:

- Entregue uma função, componente ou endpoint por prompt.
- Verifique se cada peça funciona antes de seguir adiante.
- Construa sobre código confirmado como funcional, não sobre uma torre de saída não verificada.

```
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 menores significam diffs menores, e diffs menores significam que você consegue de fato ler o que mudou. Esse é o jogo inteiro. Há um segundo ganho também: quando algo quebra, um passo pequeno é trivial de isolar. Se você gerou e verificou o endpoint de cadastro, depois a lógica de sessão, depois a redefinição de senha, um bug que aparece após o passo três quase certamente vive no passo três. A geração de uma vez só joga fora esse sinal — o bug pode estar em qualquer lugar entre quatrocentas linhas que você nunca leu.

Peça um Plano Antes do Código

Para qualquer coisa não trivial, separe *pensar* de *digitar*. Peça à IA para esboçar sua abordagem primeiro. Isso pega suposições erradas antes que elas virem 200 linhas de código errado, e é muito mais barato consertar um plano do que um pull request.

```
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.
```

Quando o plano parece certo, diga “good, implement step 1”. Quando não parece, você gastou trinta segundos corrigindo o rumo em vez de uma hora desembranando código. Planejar primeiro é o hábito que mais separa o prompting no estilo de engenheiro do prompting baseado em desejo.

A etapa de planejamento também é onde você pega a IA escolhendo uma solução mais pesada do que você precisa — recorrendo ao Redis quando um mapa em memória resolveria, ou reinventando à mão algo que uma chamada de biblioteca de uma linha já resolve. Leia o plano com uma pergunta em mente: *isso é a coisa mais simples que poderia funcionar?* Se a abordagem cheirar a superengenharia, questione com

palavras simples (“this is a hobby project with one server — can we skip the distributed store?”) antes que uma única linha seja escrita.

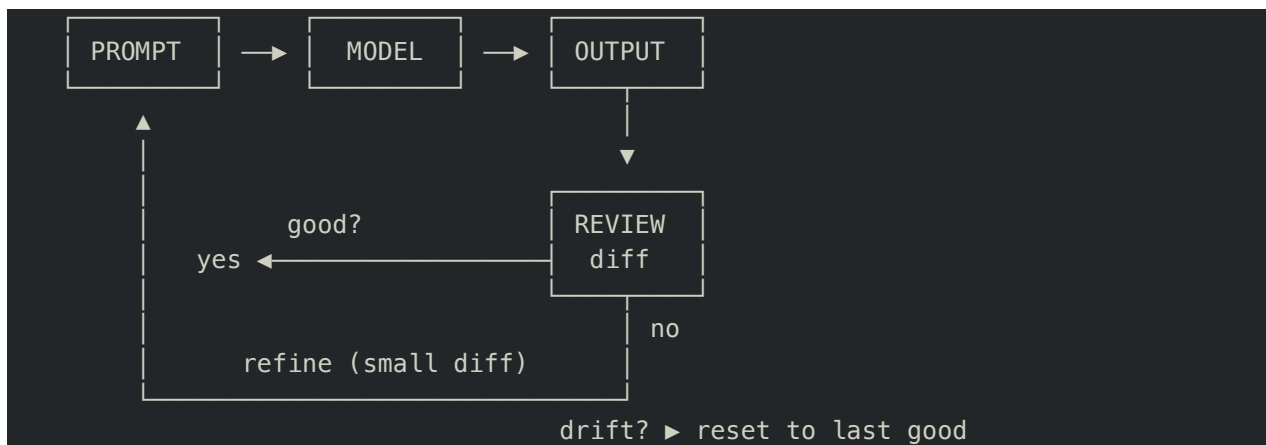
Itere em Diffs, Não em Reescritas

Uma vez que o código existe, resista ao impulso de pedir tudo de novo. Follow-ups vagos como “make it better” jogam fora código funcional e reintroduzem bugs que você já tinha corrigido. Aponte para a mudança específica que você quer.

```
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.
```

O ciclo saudável é um loop apertado: dê o prompt ao modelo, leia a saída e depois refine com um diff pequeno e direcionado — nunca uma reescrita completa — até ficar certo:



Pedir um diff (em vez de uma reescrita completa) mantém as mudanças revisáveis e preserva as partes que já funcionam. Trate a saída da IA como o pull request de uma colega: comente linhas específicas, peça edições direcionadas. Quando uma mudança sai do rumo e o código se afasta cada vez mais do que você queria a cada resposta, não continue remendando — volte para a última versão em que você confiava e reformule o prompt a partir dali com uma descrição mais afiada. Iterar para a frente sobre uma base quebrada só piora a bagunça.

Forneça Exemplos

Um exemplo concreto vale um parágrafo de descrição. Se você quer a saída em um formato específico, mostre o formato. Essa é a forma mais rápida de eliminar ambiguidade.

```
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.

Exemplos fixam nomenclatura, estrutura e capitalização que as palavras sozinhas deixam vagas. Quando possível, inclua o exemplo como um teste — aí “make the test pass” vira uma instrução inequívoca. O truque se generaliza: uma resposta de amostra de uma API (Application Programming Interface — Interface de Programação de Aplicações, o jeito como um programa pede dados a outro) fixa um parser, um par antes/depois fixa uma refatoração, uma captura de tela ou um esboço em ASCII fixa um layout. Sempre que você se pegar escrevendo uma descrição longa de um formato, pare e mostre uma instância desse formato.

Diga à IA o Que Não Fazer

Restrições não são só sobre o resultado que você quer — são também sobre os movimentos que você quer que a IA evite. Sem restrições, modelos tendem a adicionar dependências, “ajudar” refatorando código que você não pediu para mexer, e inventar funcionalidades nas bordas do seu pedido. Algumas restrições negativas mantêm a saída enxuta.

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

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

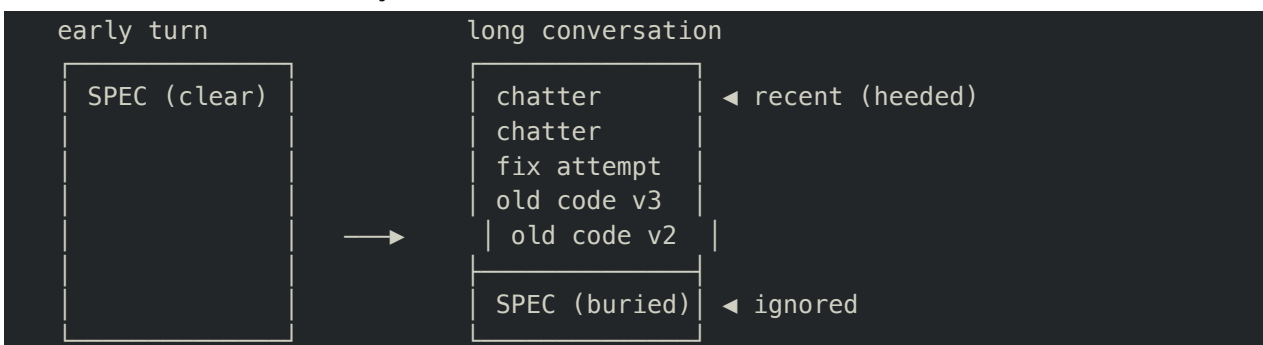
Constraints:

- 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.

O segundo prompt bloqueia de antemão uma dúzia de formas pelas quais o primeiro poderia dar errado. Restrições negativas são especialmente valiosas em uma base de código existente, onde o custo de uma mudança não solicitada não é só revisá-la — é o conflito de merge, a colega surpresa e a regressão em um código que estava funcionando bem. Se há uma linha que você não quer que seja cruzada, diga isso explicitamente; a IA não vai inferir seus limites por educação.

Gerencie o Contexto Como um Recurso

Uma conversa longa não é de graça. À medida que uma thread cresce, instruções anteriores ficam enterradas, o modelo se apoia mais nas mensagens recentes, e detalhes que você declarou vinte turnos atrás silenciosamente deixam de importar. Imagine a janela de contexto como uma caixa de tamanho fixo: conforme a conversa cresce, ela se enche, e suas instruções mais antigas são empurradas para o fundo, onde o modelo dá menos atenção a elas:



XXXXXXXXXXXX

◀ pushed out

Trate a janela de contexto como uma superfície de trabalho pela qual você é responsável por manter limpa, não como uma memória infinita.

Alguns hábitos ajudam:

- **Reformule a especificação quando você mudar de direção.** Não assuma que um requisito de dez mensagens atrás ainda vale. Resuma o alvo atual em um parágrafo curto e deixe a IA trabalhar a partir dele.
- **Comece do zero para um novo problema.** Quando você mudar do código de autenticação para um bug de CSS (Cascading Style Sheets — Folhas de Estilo em Cascata, a linguagem que controla a aparência de uma página), abra uma nova conversa. Arrastar contexto não relacionado piora o modelo, não melhora.
- **Cole o código atual, não o histórico.** Se uma função mudou cinco vezes, a IA não precisa das cinco versões antigas — ela precisa da que existe agora. Dê a ela o estado presente e a próxima mudança.
- **Mantenha uma fonte de verdade curta fora do chat.** Para uma tarefa mais longa, algumas linhas em um arquivo (“stack, decisões-chave, o que está pronto, o que vem a seguir”) permitem que você reinicie uma conversa nova em segundos, em vez de reconstruí-la a partir da rolagem de mensagens.

O objetivo é o mesmo que percorre este capítulo inteiro: a cada momento, a IA deve estar trabalhando a partir de uma imagem precisa e compacta da tarefa — não de uma transcrição extensa pela qual ela tem que adivinhar seu caminho.

Antipadrões Comuns a Evitar

Estes são os hábitos de prompting que silenciosamente desperdiçam seu tempo. Fique atento a eles nas suas próprias sessões.

- **O desejo vago.** “Make a good login page.” Bom segundo qual padrão? Substitua adjetivos por especificidades: quais campos, qual validação, qual framework.
- **O megaprompt.** Despejar uma funcionalidade inteira em uma única mensagem. Divida em passos que você consiga verificar um de cada vez.
- **O contexto silencioso.** Assumir que a IA lembra ou conhece seu projeto. Reafirme os arquivos, tipos e convenções relevantes — especialmente em uma conversa nova.
- **A aceitação cega.** Colar código gerado sem lê-lo. A IA é confiante mesmo quando está errada; você ainda é a engenheira responsável.
- **A repetição infinita.** Rodar “try again” repetidamente esperando uma saída diferente. Se duas tentativas falharem, o prompt é o problema — adicione contexto ou restrições, não apenas jogue de novo.
- **O alvo móvel.** Mudar requisitos no meio da thread sem reafirmá-los. Resuma a especificação atual quando você mudar de direção, para que a IA não esteja trabalhando a partir de uma imagem desatualizada.
- **O loop de bajulação.** Aceitar “You’re absolutely right!” como prova de que o conserto está correto. Concordância não é verificação — rode o código e confira o comportamento você mesma.

A Mentalidade Central

Toda técnica aqui se reduz a uma ideia: **diga o que você teria que dizer a uma colega competente que não consegue ler sua mente.** Contexto no início. Entradas e saídas claras. Passos pequenos e verificáveis. Um plano antes do código. Edições direcionadas em vez de reescritas. Exemplos onde as palavras não bastam. Limites explícitos sobre o que não tocar. Uma imagem limpa e atual da tarefa em todos os momentos.

Você não está escrevendo encantamentos mágicos. Está escrevendo uma especificação, de forma breve e precisa, e depois revisando o resultado. Faça isso de forma consistente e a IA para de parecer uma máquina caça-níqueis e passa a parecer a engenheira rápida e incansável que ela pode de fato ser — e, tão importante quanto isso, você continua sendo a pessoa que entende o que foi entregue.

Recapitulação e Prática

Principais conclusões

- Dê contexto antes da tarefa: declare a stack, as convenções e as peças existentes a reutilizar — ou cole a assinatura real com a qual o código precisa se encaixar.
- Defina os limites: entradas, saídas e casos de borda. O comportamento interessante do software vive nas bordas, então gaste suas palavras ali.
- Trabalhe em passos pequenos e verificáveis e peça um plano antes do código; é muito mais barato consertar um plano do que um pull request.
- Itere em diffs direcionados, não em reescritas completas, e use exemplos concretos (idealmente como testes) para fixar qualquer coisa que as palavras deixem vaga.
- Gerencie a janela de contexto como um recurso, e diga o que não fazer — restrições negativas explícitas mantêm a saída enxuta.

Coloque em prática

Pegue um pedido vago de uma linha que você normalmente digitaria (“add search to the list”) e o reescreva como um prompt de engenheira antes de enviar qualquer coisa. Adicione o contexto (stack, o arquivo que ele afeta), as entradas/saídas, dois ou três casos de borda, um exemplo do resultado desejado, e uma restrição negativa (o que não tocar). Envie a versão reescrita e compare com o que a versão preguiçosa teria produzido. Faça isso para três pedidos reais e o upgrade se torna automático.

Prompt do 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>.
```

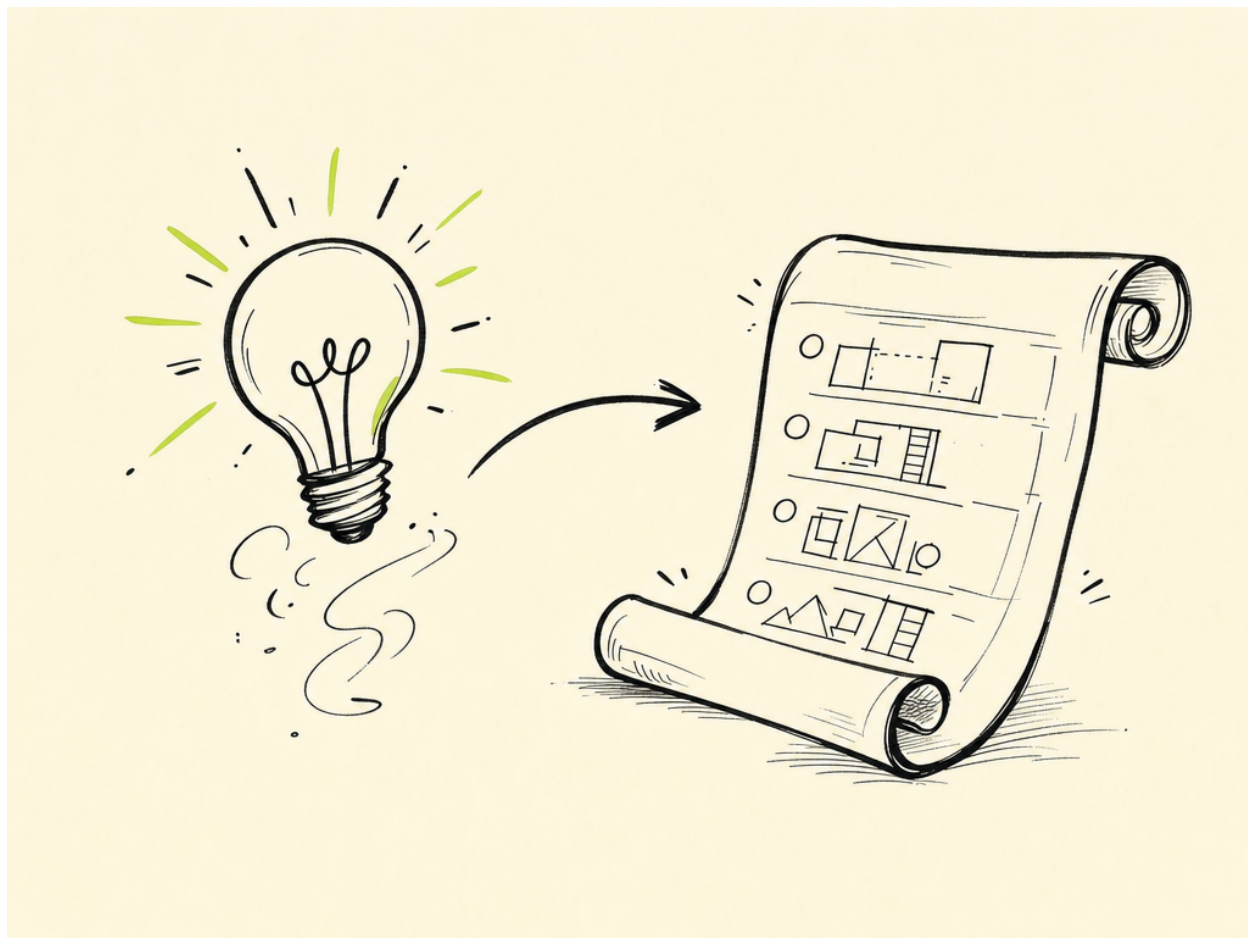
```
Example: <one concrete input -> expected output>.
```

```
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".
```


Da Ideia à Especificação em Minutos



Todo projeto começa como um pensamento vago: “quero um app que me ajude a acompanhar minhas leituras.” É uma boa semente, mas você não consegue construir a partir dela diretamente — e sua IA também não. Uma ideia vaga entregue a uma IA produz software vago: recursos que você não pediu, partes que faltam que você queria, e uma pilha de código que você não entende.

A solução é uma **especificação** (spec) — um documento curto, em linguagem simples, que diz o que você está construindo e o que você *não* está construindo. A boa notícia: com a IA como parceira de raciocínio, você consegue ir de uma ideia de uma linha a uma especificação pronta para construir em minutos, não em dias. Este capítulo mostra como.

Para deixar claro o que uma spec *não* é: não é um contrato, não é um documento de design que você vai defender numa reunião, e não é algo que você escreve uma vez e congela. É uma anotação de trabalho para você e para sua IA. Você vai editá-la conforme aprende. O objetivo não é um documento perfeito — é um documento *bom o suficiente* que deixa a próxima hora de construção mais clara do que a anterior.

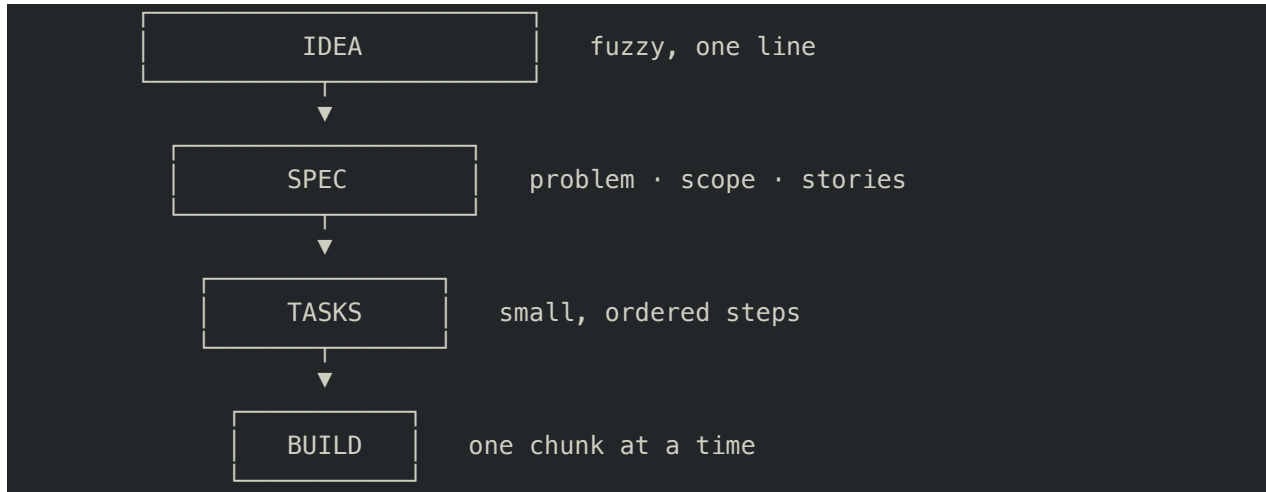
Por Que uma Spec, Não Só Vibração

Vibe coding não significa pular o planejamento. Significa planejar *rápido* e *de forma leve*, e então direcionar a IA para executar. Uma spec é o volante. Sem ela, a IA chuta sua intenção a cada prompt, e pequenos mal-entendidos se acumulam numa bagunça.

Uma spec te dá três coisas:

- **Contexto compartilhado** — você pode colar a spec em qualquer sessão de IA e reestabelecer instantaneamente o que está construindo.
- **Um limite de escopo** — uma linha escrita entre “dentro” e “fora” que impede o crescimento descontrolado de recursos (feature creep).
- **Uma lista de tarefas** — pedaços construíveis que a IA pode completar um de cada vez.

Uma spec funciona como um funil: pega uma ideia ampla e vaga e a estreita, passo a passo, em pequenos pedaços que a IA consegue de fato construir.



Ela não precisa ser longa. Uma página basta para a maioria dos projetos. Na verdade, uma spec que passa de duas páginas geralmente é um sinal de alerta: você está planejando longe demais ou expandindo o escopo silenciosamente. Na dúvida, corte. As partes que você apaga raramente fazem falta, e as partes que você mantém ficam mais afiadas.

Passo 1: Esclareça o Problema

Antes dos recursos, nomeie o problema e a pessoa que o tem. Resista à tentação de pular direto para soluções. Uma declaração de problema clara mantém toda decisão posterior honesta.

Pergunte a si mesmo (ou peça para a IA te entrevistar):

- Para quem é isso? (Seja específico — “eu”, “makers independentes”, “meu clube de corrida.”)
- O que essa pessoa faz hoje, de forma dolorosa, sem isso?
- Como é o sucesso, em uma frase?

Você pode deixar a IA fazer essa escavação. Experimente este prompt:

```

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 a cinco boas perguntas geralmente transforma uma ideia vaga em algo real. O truque é deixar a IA questionar em vez de concordar com tudo. Se ela perguntar “você precisa de sincronização entre dispositivos?” e sua resposta honesta for “não, só o meu laptop”, você acabou de eliminar um backend, um sistema de autenticação e semanas de trabalho — antes de escrever uma linha de código. Esse é o ponto principal: o lugar mais barato para cortar um recurso é numa conversa, não numa base de código.

Fique atento ao momento em que suas respostas deixam de ser “ah, acho que sim” e passam a ser “sim, exatamente isso.” Essa mudança é o sinal de que você encontrou o problema real e pode seguir em frente.

Passo 2: Defina o Escopo e os Não-Objetivos

Este é o passo mais valioso e o mais pulado. **Não-objetivos** (non-goals) — as coisas que você deliberadamente *não* vai construir — te protegem de uma IA que adora adicionar extras “úteis”.

Para nosso rastreador de livros, o escopo poderia ser:

- **Dentro do escopo:** adicionar um livro, marcá-lo como lendo/terminado, ver uma lista, avaliar livros terminados.
- **Fora do escopo (não-objetivos):** recursos sociais, recomendações, importação do Goodreads, app mobile, contas de usuário.

Imagine duas colunas: tudo que você se compromete a construir à esquerda, tudo que você deliberadamente pula à direita. A linha entre elas é a fronteira que mantém a v1 pequena.

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

Escrever os não-objetivos cumpre duas funções. Mantém sua primeira versão pequena o suficiente para de fato terminar, e te dá uma frase para colar quando a IA se desviar: *“Lembre-se, contas são um não-objetivo para a v1 — não adicione login.”*

Um não-objetivo não é “nunca” — é “não agora.” Listar contas de usuário como não-objetivo não significa que seu app nunca as terá. Significa que elas não fazem parte *desta* versão, então você não paga o custo delas hoje. Esse reenquadramento importa: ele te permite dizer não a um recurso sem sentir que está abandonando uma boa ideia. Você está apenas sequenciando. A maioria dos itens da sua lista de não-objetivos é, na verdade, um backlog de v2 disfarçado, e tudo bem — anote em algum lugar e siga em frente.

Passo 3: Escreva Histórias de Usuário

Histórias de usuário (user stories) transformam escopo em comportamento testável. O formato é simples:

Como **[usuário]**, eu quero **[ação]**, para que **[benefício]**.

Para o rastreador de livros:

- Como leitor, eu quero adicionar um livro por título e autor, para que eu possa começar a acompanhá-lo.
- Como leitor, eu quero marcar um livro como terminado, para que eu possa ver meu progresso.
- Como leitor, eu quero avaliar um livro terminado de 1 a 5 estrelas, para que eu lembre do que gostei.

Cada história é uma unidade que você pode construir e verificar de forma independente. Se você não consegue imaginar como testaria uma história clicando pelo app, ela está vaga demais — divida-a ou reescreva-a.

A cláusula “para que” é a parte que as pessoas deixam de lado, e é a mais importante. “Como leitor, eu quero uma caixa de busca” diz à IA o *quê* construir, mas não *por quê*, então ela não consegue fazer boas escolhas de compromisso. “Como leitor, eu quero encontrar um livro na minha lista digitando parte do título, para que eu não precise rolar por dezenas de itens” diz a ela o trabalho de fato — e agora um simples filtro-enquanto-digita pode superar um mecanismo de busca pesado. O benefício é a spec da

spec: ele explica o recurso bem o suficiente para que a IA (e o você do futuro) consiga distinguir uma boa implementação de uma errada.

Passo 4: Rascunhe um PRD Leve

Um PRD (Product Requirements Document, ou Documento de Requisitos de Produto) é apenas suas decisões reunidas num só lugar. Ele tem um esqueleto previsível — as mesmas poucas seções toda 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
```

Mantenha-o pequeno. Aqui está um PRD inicial completo que você poderia entregar a uma 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.
```

Repare na linha **“Done When”** (“Pronto Quando”). Esse é seu teste de aceitação — uma forma concreta de saber que você entregou a v1. Sempre inclua uma. Um bom “Done When” é algo que você consegue *executar fisicamente*: clicar aqui, digitar isso, atualizar a página, ver aquilo. Se sua condição de pronto soa como um sentimento (“quando estiver polido”, “quando parecer bom”), ela nunca vai estar pronta, porque

não há nada contra o que checar. Torne-a um roteiro que você poderia entregar a um estranho.

Você não precisa escrever o PRD à mão. Um movimento útil é alimentar a IA com as respostas dos passos anteriores e deixá-la montar o rascunho, e então você edita. O ponto de editar em vez de aceitar tudo de uma vez é que você precisa entender e concordar com cada linha — este documento é a fonte da verdade que você vai continuar colando de volta, então um erro aqui é copiado para toda sessão seguinte.

Passo 5: Esboce o Modelo de Dados

A maioria dos apps é, na verdade, sobre *dados*: o que você guarda e como as partes se relacionam. Um esboço rápido do modelo de dados te poupa de código confuso e inconsistente mais tarde. Você não precisa de um diagrama de banco de dados — uma lista de campos já basta.

Para o rastreador de livros, há uma coisa principal — um Book (Livro):

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

Mesmo este esboço minúsculo torna as decisões explícitas: avaliações só existem para livros terminados, o status é um de três valores fixos, e todo livro tem um id. Quando você entrega isso à IA, ela para de adivinhar nomes de campos e constrói de forma consistente.

O motivo para fixar isso cedo é que nomes de campos vazam por todo lugar. Assim que a IA escreve código que lê `book.rating`, esse nome aparece no formulário, na visualização de lista, na camada de armazenamento, e em qualquer recurso futuro que você adicionar. Se você deixar esse nome variar — `rating` em um lugar, `stars` em outro, `score` em um terceiro — você tem bugs cansativos de rastrear. Um modelo de dados de seis linhas é o seguro mais barato contra isso.

Um bloco de spec curto e pronto para copiar e colar geralmente funciona melhor do que prosa para a IA. Aqui está o mesmo modelo escrito como um prompt que você de fato enviaria:

```
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.
```

Essa última linha — *pergunte a mim antes de adicionar* — está fazendo um trabalho real. Ela transforma a IA de uma adivinhadora numa colaboradora e mantém o modelo de dados seu.

Passo 6: Divida a Spec em Tarefas do Tamanho de uma Vibe

O movimento final é fatiar sua spec em tarefas que a IA consegue executar **uma de cada vez**. Uma tarefa do tamanho de uma vibe é pequena o suficiente para você revisar o resultado em um minuto e saber se

funcionou. Se uma tarefa exige vários arquivos da IA e você não consegue dizer se está certa, ela está grande demais.

Boas tarefas para o rastreador de livros:

1. Montar o layout básico da página com um cabeçalho e uma lista de livros vazia.
2. Construir o formulário “adicionar um livro” (título + autor) que anexa à lista em memória.
3. Adicionar status (Para Ler / Lendo / Terminado) com um menu suspenso para mudá-lo.
4. Salvar a lista no armazenamento local do navegador para que sobreviva a uma atualização de página.
5. Adicionar um controle de avaliação que aparece só para livros terminados.
6. Adicionar botões de filtro de status acima da lista.

Cada tarefa mapeia para uma história de usuário ou linha do PRD, constrói sobre a anterior, e termina em algo que você consegue *ver* funcionando. Esse é o ritmo do *vibe coding*: tarefa pequena, executar, verificar, próxima.

Você pode até pedir para a IA gerar essa divisão para você:

```
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.
```

Mantenha a Spec Viva

A spec não é um monumento que você constrói uma vez e admira. É um documento vivo, e o erro mais comum é tratá-la como fixa. Enquanto você constrói, vai aprender coisas que a fase de planejamento não conseguia te contar: um recurso é mais difícil do que o esperado, um não-objetivo acaba sendo essencial, uma história de usuário era secretamente duas histórias. Quando isso acontecer, edite a spec *primeiro*, e depois diga à IA para seguir a nova versão. A spec lidera; o código segue.

Alguns hábitos mantêm isso honesto:

- **Atualize-a no momento em que uma decisão muda.** Uma spec que diz uma coisa enquanto seu código faz outra é pior do que nenhuma spec — ela ativamente engana a próxima sessão de IA.
- **Risque tarefas conforme as termina.** A lista de tarefas funciona também como rastreador de progresso. Ver seis de oito itens marcados já é um tipo de impulso.
- **Mova surpresas para os não-objetivos ou para uma lista “depois”** em vez de simplesmente fazê-las. Se um recurso tentador aparecer no meio da construção, anote-o e continue. Você pode decidir depois se ele merece um lugar na v1.

Essa é a disciplina que separa “eu construí uma coisa” de “eu construí a coisa que eu queria construir.” A spec é como você continua sendo o autor do seu projeto, em vez de um passageiro nele.

Uma Armadilha Comum: Especificar Soluções, Não Problemas

Um modo de falha vale a pena destacar sozinho, porque é muito fácil de cair nele. Quando você senta para escrever uma spec, a tentação é descrever a *solução* que você já imaginou — “uma barra lateral com abas”, “um modal que aparece”, “uma tabela do Postgres.” Mas uma spec cheia de soluções trava silenciosamente decisões que você ainda não ganhou o direito de tomar.

Compare estas duas linhas:

```
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.
```

A primeira frase já escolheu uma barra lateral, seções recolhíveis e um layout — antes de qualquer um confirmar que essas são as escolhas certas. A segunda declara o *trabalho a ser feito* e deixa a IA (e

você) livres para encontrar a coisa mais simples que o satisfaça, que pode ser apenas três botões de filtro. Descreva o resultado que você quer; deixe a implementação negociável pelo máximo de tempo possível. Soluções são baratas de mudar numa spec e caras de mudar em código.

A Recompensa

Agora você tem uma declaração de problema, escopo e não-objetivos, histórias de usuário, um PRD de uma página, um modelo de dados, e uma lista de tarefas ordenada — a fundação inteira de um projeto real, rascunhada em minutos com a IA como sua caixa de ressonância.

Nada disso exigiu que você soubesse programar. Exigiu que você soubesse o que queria e pensasse com clareza por alguns minutos — e a IA cuidou do resto do raciocínio em voz alta com você. Essa é a verdadeira mudança: você ainda não está escrevendo software, você está escrevendo a *intenção*, e boa intenção é a maior parte da batalha.

Mantenha essa spec num arquivo dentro do seu projeto (muita gente usa um `SPEC.md` ou `PRD.md`). Ela se torna o documento que você cola em sessões novas de IA para reancorar o contexto, e a lista de verificação que você percorre uma tarefa do tamanho de uma vibe por vez. No próximo capítulo, vamos pegar a primeira tarefa e de fato construí-la.

Recapitulação e Prática

Principais aprendizados

- Uma spec é uma anotação de trabalho curta, em linguagem simples — o que você está construindo e o que *não* está — não é um contrato que você congela; uma página basta.
- Esclareça o problema e a pessoa antes dos recursos, e anote os não-objetivos: eles são sua defesa mais afiada contra uma IA que adiciona extras “úteis”.
- Histórias de usuário com uma cláusula “para que” e um PRD minúsculo com uma linha “Done When” concreta transformam intenção em comportamento testável e construível.
- Fixe o modelo de dados cedo — nomes de campos vazam por todo lugar — e fatie a spec em tarefas do tamanho de uma vibe que você consegue verificar uma de cada vez.
- Descreva o problema, não a solução; resultados continuam baratos de mudar numa spec e caros de mudar em código. Mantenha a spec viva conforme você aprende.

Experimente

Pegue qualquer ideia que você vinha guardando e transforme-a numa spec de uma página em uma única sessão de IA. Deixe a IA te entrevistar com cinco perguntas afiadas primeiro, e então escreva — com suas próprias palavras — o problema, o usuário, três histórias de usuário (cada uma com um “para que”), uma lista explícita de não-objetivos, um modelo de dados de seis linhas, e uma linha “Done When” que você conseguiria executar fisicamente. Salve como `SPEC.md`. O teste de uma boa spec: você conseguiria colá-la num chat novo amanhã e a IA saberia exatamente o que construir.

Prompt do 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.
```

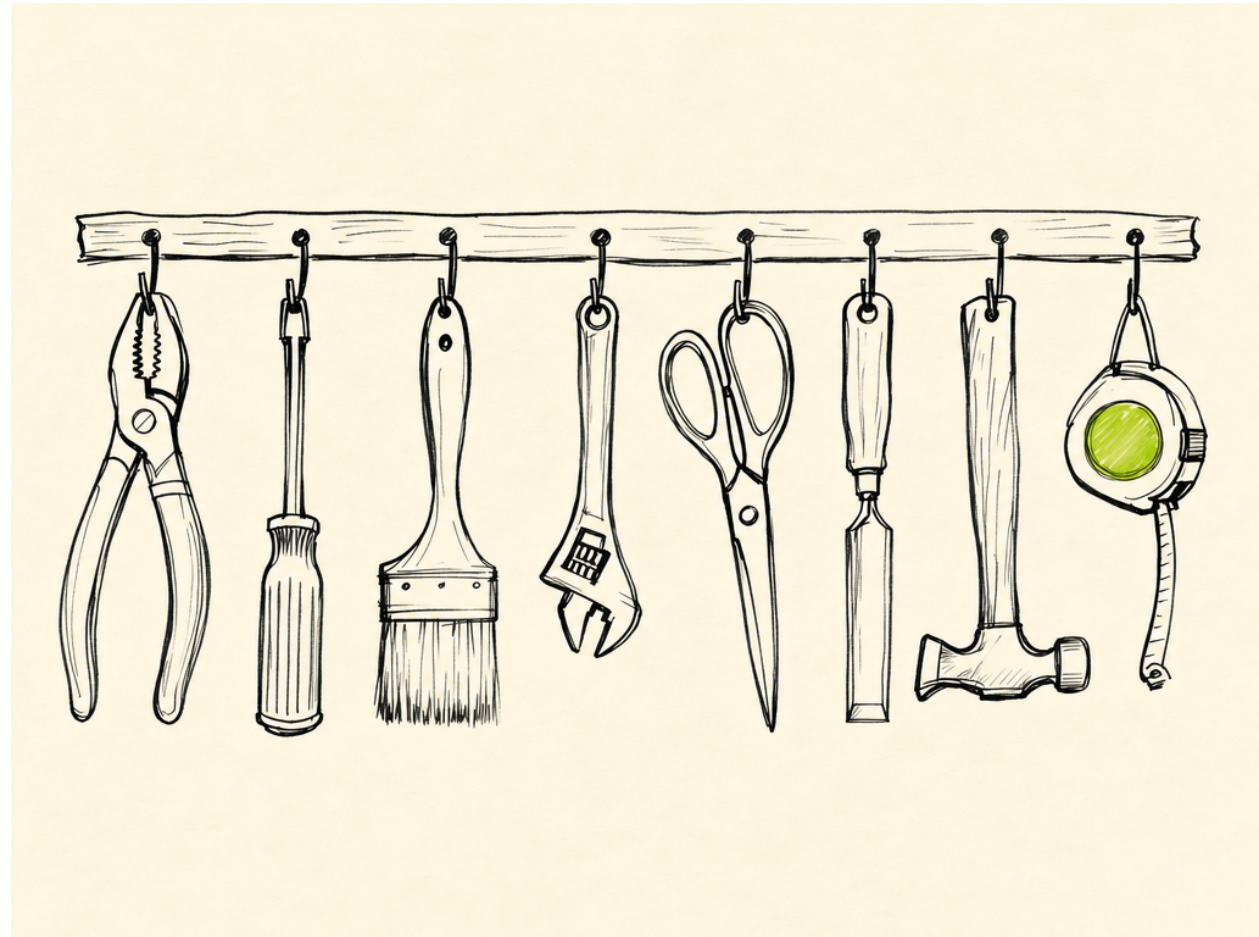
```
After I answer, draft a one-page spec with these sections:
```

- 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.

Escolhendo Sua Stack: Linguagens em Perspectiva

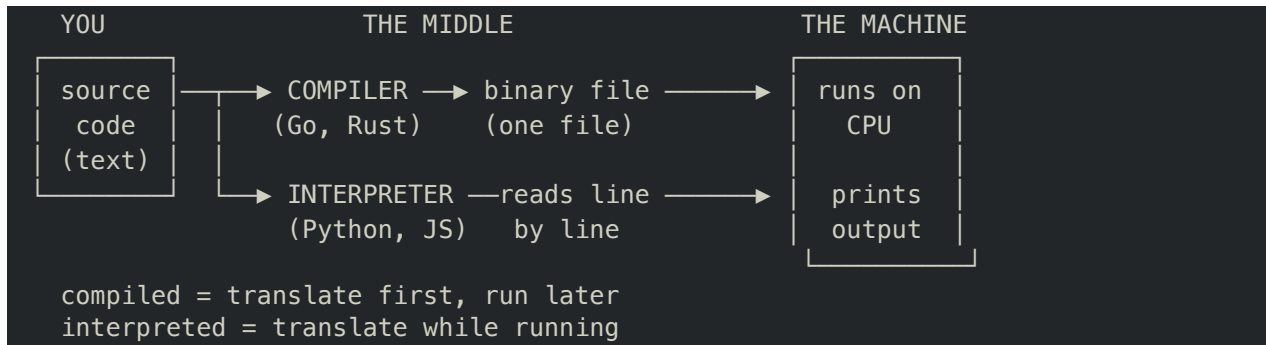


Quando você direciona uma IA para construir software, a linguagem que você escolhe ainda importa — e muito. A IA escreve o código, mas você vive com as consequências: com que velocidade você entrega, quão fácil é fazer o deploy, com que frequência você esbarra numa parede. A boa notícia é que você não precisa dominar nenhuma dessas linguagens. Você precisa saber o suficiente para fazer uma escolha sensata e para reconhecer quando a IA está te levando para um caminho doloroso.

Este capítulo percorre as linguagens que valem a pena conhecer, no que cada uma é genuinamente boa, onde ela morde, e — crucial para vibe coding — quão bem os assistentes de IA lidam com ela. Vamos terminar com um guia simples de escolha por objetivo.

Uma observação antes de começarmos: os assistentes de IA são *muito* melhores em linguagens populares com montanhas de código público do que em linguagens de nicho. Esse não é um efeito pequeno. É frequentemente o único maior fator em se o seu projeto vai bem ou não. Popularidade é um recurso. Quando uma linguagem e seus principais frameworks aparecem milhões de vezes em repositórios públicos, a IA já viu todo padrão comum, todo erro típico e toda correção idiomática. Quando não aparecem, você paga por essa escassez em tentativas fracassadas e bobagens ditas com confiança.

Antes dos nomes das linguagens, uma imagem que vale a pena guardar na cabeça: o código que você escreve nunca é o que a máquina de fato executa. Algo no meio transforma seu texto em software em execução — ou um *compilador* (traduz tudo antecipadamente em um arquivo que você executa) ou um *interpretador* (lê e executa linha por linha). A linguagem que você escolhe decide em qual caminho você está.



JavaScript / TypeScript

Esta é a escolha padrão para a maior parte do vibe coding, e por um bom motivo. Roda em qualquer lugar — navegadores, servidores (Node.js), funções de borda (edge functions) — e é a linguagem de todo o frontend da web. TypeScript é JavaScript com um sistema de tipos acoplado, e você quase sempre deve preferir TypeScript: os tipos capturam erros cedo e, o que é importante, dão à IA balizas que tornam sua saída mais confiável. Uma assinatura de função tipada é um contrato que o assistente consegue ler; sem ele, a IA está adivinhando qual formato seus dados têm, e adivinhar é onde os bugs se escondem.

- **Boa para:** aplicações web (frontend e backend), qualquer coisa em que o usuário clica em um navegador, protótipos rápidos, projetos full-stack em uma única linguagem, deploy serverless/edge.
- **Onde dói:** o ecossistema se move rápido e quebra coisas; a proliferação de dependências é real (um app pequeno pode puxar centenas de pacotes); a liberdade do JavaScript puro deixa bugs sutis escaparem — outro motivo para usar TypeScript.
- **Casos de uso típicos:** dashboards de SaaS (Software as a Service — software que você aluga online em vez de instalar), landing pages, e-commerce, aplicações em tempo real, extensões de navegador, qualquer coisa que você faça deploy na Vercel, Netlify ou Cloudflare.
- **Manejo pela IA:** excelente. Esta é a linguagem mais representada nos dados de treinamento. Assistentes escrevem React, Next.js e Node com confiança e raramente travam. Se você não tem um motivo forte para escolher outra coisa, comece por aqui.

Uma troca concreta que vale a pena conhecer: a velocidade que torna o JavaScript ótimo para prototipagem é a mesma velocidade que o faz mudar constantemente. Um tutorial de dezoito meses atrás pode já usar uma API obsoleta, e a IA — treinada sobre uma fotografia do passado — às vezes vai te entregar o padrão de ontem. Quando algo que ela escreve não funciona, a causa costuma ser que a biblioteca mudou. A correção geralmente é dizer a ela a versão que você está usando e pedir que verifique a documentação atual. Este é um imposto que você paga por viver no ecossistema mais ativo do planeta, e para a maioria dos projetos vale a pena pagá-lo.

Python

Python se lê quase como inglês, o que o torna a linguagem mais amigável para iniciantes e a língua franca de dados, scripting e da própria IA. Se o seu projeto toca aprendizado de máquina, análise de dados, automação ou computação científica, Python costuma ser a resposta.

- **Bom para:** IA/ML, processamento de dados, scripts de automação, APIs de backend (uma API, ou Application Programming Interface — Interface de Programação de Aplicações —, é a porta de

entrada que um programa usa para conversar com outro), código de “cola” (glue code), qualquer coisa em que já existam bibliotecas prontas (e para Python, geralmente existem).

- **Onde dói:** lento em processamento numérico bruto, a menos que você se apoie em bibliotecas escritas em C por baixo dos panos; o “inferno das dependências” com ambientes virtuais e versões de pacotes confunde iniciantes; não serve para frontends de navegador.
- **Casos de uso típicos:** scraping, pipelines de dados, experimentos de ML, bots de Discord/Telegram, ferramentas internas, APIs REST (Representational State Transfer — Transferência de Estado Representacional, um estilo comum para essas portas de API) (FastAPI, Django, Flask).
- **Manejo pela IA:** excelente, no mesmo nível do JavaScript. Assistentes são extremamente fluentes em Python e suas principais bibliotecas. Uma ótima escolha padrão para projetos que não são web.

O único lugar em que Python costuma tropeçar iniciantes é o ambiente, não a linguagem. Um script que roda bem para a IA em uma pasta falha em outra porque os pacotes vivem em um ambiente virtual que não foi ativado, ou porque dois projetos querem versões diferentes da mesma biblioteca. Nada disso é culpa do seu código, e nada disso é difícil uma vez que você já viu — mas é o momento mais comum de “por que isso simplesmente não roda” em Python. Peça à IA para configurar um ambiente virtual e um `requirements.txt` (ou use uma ferramenta como `uv` ou `Poetry`) desde o início, e a maior parte da dor desaparece.

Go

Go foi projetado no Google para ser simples, rápido e fácil de fazer deploy. Ele compila para um único binário autocontido que você pode colocar em um servidor sem precisar instalar nenhum runtime. Deliberadamente deixa de fora recursos sofisticados, o que mantém as bases de código “chatas” da melhor forma possível — e “chato” é fácil de raciocinar tanto para humanos quanto para IA.

- **Bom para:** serviços de backend rápidos, ferramentas de linha de comando, redes, qualquer coisa que precise lidar com muitas conexões concorrentes, deploy simples.
- **Onde dói:** tratamento de erros verboso (você vai ver `if err != nil` por toda parte); não é uma linguagem de frontend; a simplicidade deliberada pode parecer limitante se você quer atalhos expressivos.
- **Casos de uso típicos:** APIs, microsserviços, CLIs (uma CLI, ou Command-Line Interface — Interface de Linha de Comando —, é uma forma somente de texto de rodar programas digitando comandos), ferramentas de infraestrutura, backends de alto tráfego.
- **Manejo pela IA:** muito bom. O conjunto de recursos pequeno e consistente de Go significa que existe uma “forma certa” de fazer a maioria das coisas, e os assistentes a seguem bem. Menos dados de treinamento que JS/Python, mas a simplicidade compensa.

O recurso matador de Go para quem constrói sozinho é esse binário único. Não existe a dança de “instalar Node e esses quarenta pacotes no servidor”, nem incompatibilidade de versão de runtime entre a sua máquina e a produção. Você compila, copia um arquivo, executa. Só isso já torna o deploy dramaticamente menos propenso a erros — o que importa muito quando você está fazendo *vibe coding* e não quer depurar infraestrutura. A verbosidade é o custo: código que seriam três linhas em Python vira oito em Go. Mas essas linhas extras são previsíveis, e previsível é exatamente o que uma IA lida bem.

Rust

Rust te dá velocidade em nível de C com um compilador que se recusa a deixar você escrever categorias inteiras de bugs de memória. A troca é um compilador famosamente rigoroso e uma curva de aprendizado íngreme. Para *vibe coding*, Rust é uma faca de dois gumes: a IA escreve o código, mas o rigor de Rust significa que o compilador vai rejeitar o código com mais frequência, levando a mais ciclos de correção e ida e volta.

- **Bom para:** software crítico em desempenho, programação de sistemas, motores de jogos, WebAssembly, ferramentas de linha de comando onde velocidade importa, qualquer coisa em que correção e segurança são fundamentais.

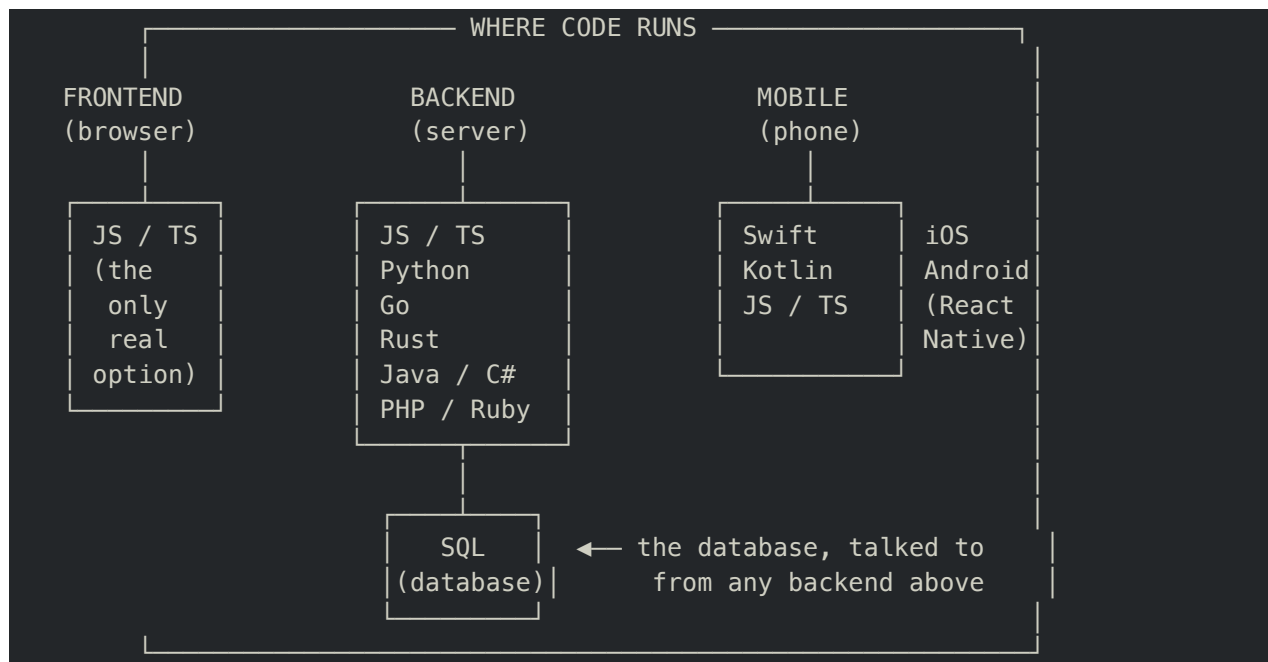
- **Onde dói:** curva de aprendizado íngreme; tempos de compilação lentos; o “borrow checker” (verificador de empréstimos) frustra iniciantes; exagero para a maioria dos aplicativos web e scripts.
- **Casos de uso típicos:** backends de alto desempenho, ferramentas CLI, módulos WASM (WebAssembly), sistemas quase embarcados, infraestrutura em que cada milissegundo conta.
- **Manejo pela IA:** bom, mas com mais solavancos. Os assistentes conhecem Rust bem, mas o compilador rigoroso expõe erros que exigem iteração. Espere mais loops de “corrigir o erro, tentar de novo” do que com Python ou JS. Escolha Rust quando desempenho ou segurança genuinamente justificarem o atrito — não por padrão.

Existe uma vantagem sutil no rigor de Rust que é fácil de deixar passar: o compilador é um revisor gratuito e implacável. Quando uma IA escreve JavaScript relapso, o bug vai para produção e morde um usuário em tempo de execução. Quando uma IA escreve Rust relapso, o compilador geralmente captura isso antes mesmo do programa rodar. Os loops extras de correção parecem atrito, mas boa parte deles é o compilador impedindo um bug real. A pergunta honesta a se fazer antes de recorrer a Rust é se você de fato precisa da velocidade ou da segurança, ou se alguém te disse que você deveria querer isso. Para um primeiro projeto, quase nunca você precisa.

Uma Palavra Rápida Sobre as Outras

Você vai esbarrar nelas, e às vezes uma delas é exatamente a certa:

Ajuda visualizar onde cada linguagem costuma viver. O software tem camadas — o que roda no navegador (frontend), o que roda em um servidor (backend), o que roda em um telefone (mobile) — e a maioria das linguagens tem um território de origem:



- **Java / C#:** os cavalos de batalha das grandes empresas. Maduras, estáveis, fortemente ferramentadas, e ideais para grandes backends corporativos e para os ecossistemas Android (Java/Kotlin) ou Windows/.NET (C#). A IA lida bem com ambas. Como um vibe coder solo, você raramente *precisa* delas, a menos que esteja trabalhando dentro de uma stack corporativa existente ou construindo especificamente para Android/Windows.
- **Swift / Kotlin:** as linguagens nativas para apps iOS (Swift) e Android (Kotlin). Se você está lançando um app mobile de verdade na App Store ou na Play Store, provavelmente vai encostar nelas. O suporte da IA é sólido, mas mais fraco que para linguagens web, e o desenvolvimento mobile carrega atrito extra (dispositivos, regras da loja de apps, ferramental de build).

- **PHP / Ruby:** a velha guarda da web, e ainda rodando silenciosamente uma fatia enorme dela. PHP alimenta o WordPress e, portanto, uma grande fração de todos os sites; Ruby on Rails continua sendo uma forma rápida e opinativa de construir um app baseado em banco de dados. Ambos têm bastante dado de treinamento, então a IA lida bem com eles — mas raramente são a primeira coisa que um recém-chegado busca hoje, e você tipicamente as escolheria por estar trabalhando em uma base de código existente, e não começando do zero.
- **SQL (Structured Query Language — Linguagem de Consulta Estruturada):** não é uma linguagem de propósito geral, mas você vai encontrá-la no momento em que seu app armazena dados. Pense nela como a linguagem que você usa para fazer perguntas a um banco de dados — “me dê todo usuário que se cadastrou esta semana”. SQL consulta bancos de dados. Você não escolhe SQL em vez das outras — ela convive ao lado de qualquer coisa que você escolher. A boa notícia: a IA é excelente escrevendo SQL, então você pode descrever em palavras simples que dados quer e deixá-la produzir a consulta.

Comparação em Perspectiva

Linguagem	Melhor para	Cuidado com	Fluência da IA
JavaScript/TypeScript	Aplicações web, full-stack, edge	Proliferação de dependências, mudança rápida	Excelente
Python	IA/ML, dados, scripts, APIs	Lento em computação bruta, configuração de ambiente	Excelente
Go	Backends rápidos, CLIs, serviços	Verboso, não serve para frontend	Muito boa
Rust	Desempenho, segurança, WASM	Curva íngreme, mais loops de correção	Boa (com solavancos)
Java/C#	Empresas, Android, .NET	Pesado, frequentemente exagero para solo	Muito boa
Swift/Kotlin	Apps nativos iOS / Android	Atrito mobile, IA mais fraca	Boa
PHP/Ruby	WordPress, Rails, apps web existentes	Padrões datados, bagagem legada	Muito boa
SQL	Conversar com bancos de dados	Convive ao lado, não em vez de	Excelente

O Custo Real de uma Escolha Exótica

É tentador escolher uma linguagem porque é elegante, ou rápida, ou porque alguém que você respeita jura por ela. Especificamente para vibe coding, esse instinto pode afundar um projeto silenciosamente. Eis o motivo: cada hora que você gasta lutando com o ferramental de uma linguagem de nicho, ou desmembrando saída de IA que está sutilmente errada porque o modelo viu poucos exemplos, é uma hora que você não gastou entregando. A escolha exótica raramente falha alto e claro. Ela falha como um gotejamento lento de loops de correção extras, bibliotecas ausentes e respostas do Stack Overflow que não existem.

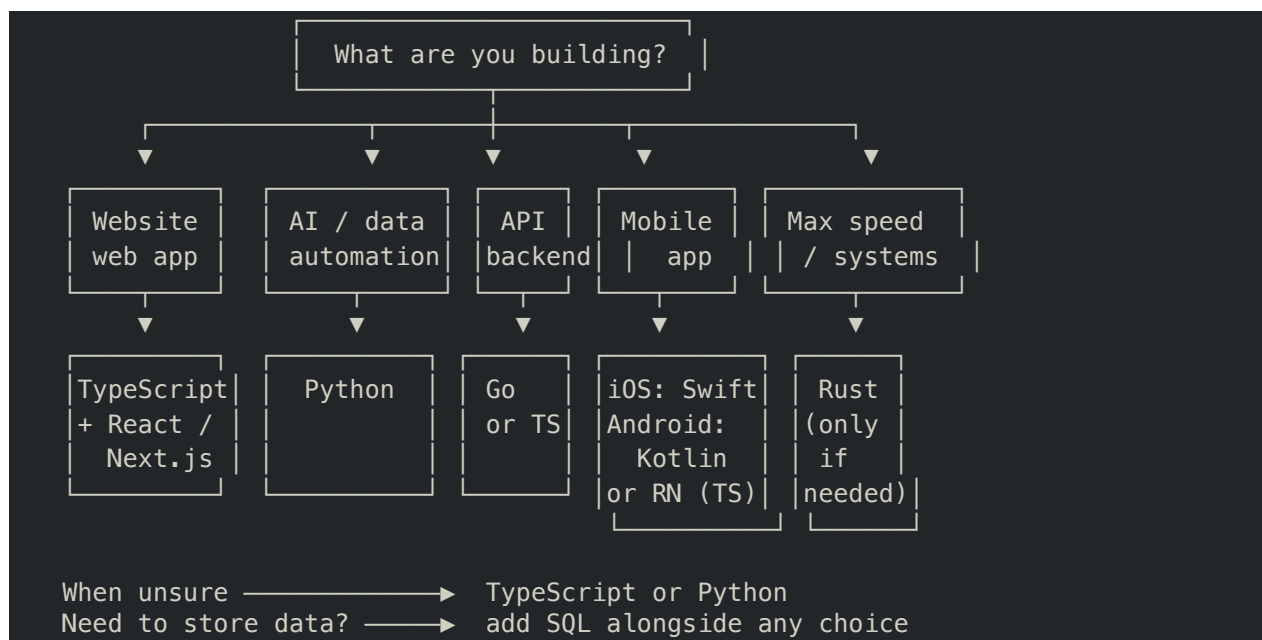
Concretamente, a penalidade aparece em três lugares. Primeiro, a IA é menos fluente, então comete mais erros e é pior em corrigi-los — e você, sem conhecer a linguagem, não consegue distinguir código bom de código ruim. Segundo, o ecossistema é mais fino, então a biblioteca de que você precisa pode não existir, pode estar abandonada, ou pode não ter exemplos para copiar. Terceiro, quando você fica travado, a web está mais silenciosa: menos tutoriais, menos perguntas respondidas, menos gente que já bateu na sua parede exata. Com Python ou TypeScript, essas três coisas simplesmente não são um problema.

Nada disso significa que linguagens exóticas são ruins. Significa que o ônus da prova recai sobre a escolha exótica. Se você tem um motivo concreto e específico — você genuinamente precisa da velocidade de Rust, é forçado a usar Swift pela App Store, está estendendo um serviço Go existente — então pague o custo de olhos abertos. Se o motivo é “parecia legal”, escolha a opção popular e entregue algo. Você sempre pode reescrever depois, e quase nunca vai fazer isso, porque a escolha popular terá cumprido o trabalho.

Escolha por Objetivo

Pule a agonia. Combine seu objetivo com uma linha e siga em frente — você sempre pode mudar de rumo depois, e para um primeiro projeto, momentum vence otimização.

Aqui está a decisão inteira como uma árvore. Comece pelo topo, responda cada pergunta, e siga o ramo até a sua stack:



- **Quero construir um site ou aplicação web.** □ TypeScript (com React/Next.js). A escolha óbvia e bem suportada.
- **Quero fazer IA, dados ou automação.** □ Python. É onde as ferramentas vivem.
- **Quero uma API rápida ou serviço de backend.** □ Go (simples e rápido) ou TypeScript (se for parte de uma stack web). O FastAPI do Python também funciona bem.
- **Estou construindo um app mobile.** □ Swift para iOS, Kotlin para Android — ou considere uma abordagem cross-platform baseada em web (React Native) para ficar em TypeScript.
- **Preciso de velocidade bruta ou controle a nível de sistema.** □ Rust, mas só se você confirmou que realmente precisa. A maioria das pessoas não precisa.
- **Estou adicionando a um site que já existe.** □ O que já estiver escrito ali. Um plugin de WordPress significa PHP; um app antigo em Rails significa Ruby. Combine com a base de código, não brigue com ela.
- **Só quero aprender e entregar algo rápido.** □ Python ou TypeScript. Ambos são tolerantes, ambos são as linguagens que a IA conhece melhor, e ambos têm tutoriais e exemplos infinitos.
- **Meu app precisa armazenar dados.** □ Qualquer que você escolha acima, mais SQL para o banco de dados. Você não escolhe entre eles.

A meta-lição: na dúvida, escolha a opção popular. TypeScript e Python cobrem a vasta maioria do que um vibe coder solo algum dia vai construir, e são as linguagens que o seu assistente de IA manuseia com mais confiabilidade. A linguagem “melhor” para vibe coding costuma ser aquela que a IA conhece melhor

— e isso significa a popular. Guarde as escolhas exóticas para quando você tiver um motivo concreto, e deixe a IA lidar com a sintaxe de qualquer forma.

Recapitulação e Prática

Principais lições

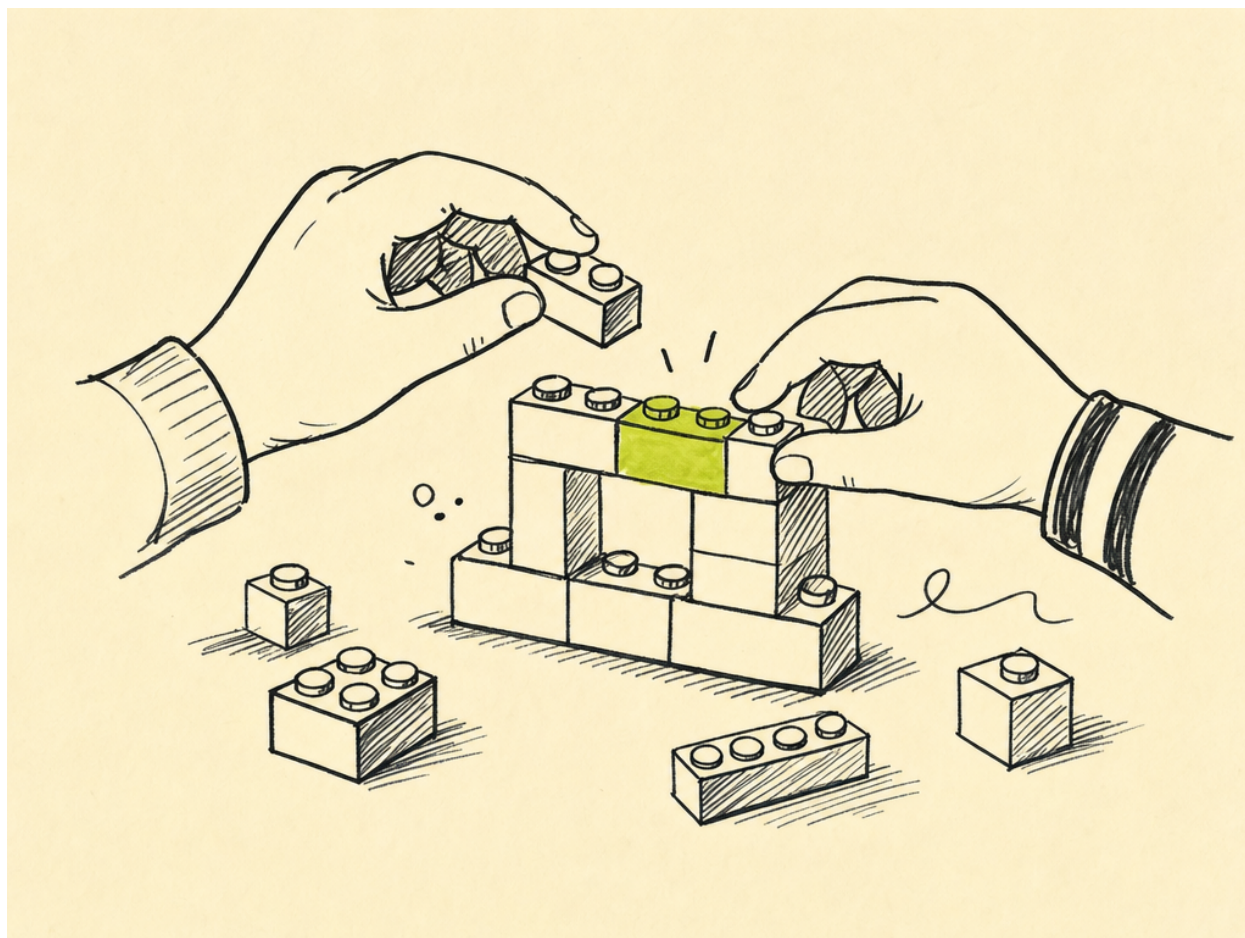
- A IA é muito mais fluente em linguagens populares do que em linguagens de nicho — popularidade é um recurso, frequentemente o único maior fator em se um projeto vai bem ou não.
- TypeScript e Python são as escolhas padrão seguras: web/full-stack para TypeScript, IA/dados/automação para Python, e ambos têm excelente suporte de IA.
- Go (backends de binário único), Rust (velocidade/segurança, mas mais loops de correção), e o resto têm cada um um ponto ideal estreito — recorra a eles só com um motivo concreto.
- SQL não é uma escolha do tipo ou-ou; ele convive ao lado de qualquer coisa que você escolher no momento em que seu app armazena dados.
- Combine a linguagem com o seu objetivo, não com o que soa impressionante — momentum vence otimização em um primeiro projeto.

Experimente

Pegue uma ideia de projeto que você realmente quer construir e escreva uma frase descrevendo seu objetivo. Combine-a com uma linha da lista “Escolha por Objetivo”, depois peça ao seu assistente de IA para confirmar ou desafiar a escolha. Peça a ele para expor a stack padrão (linguagem, framework, onde você faria o deploy) em três marcadores, e peça que nomeie uma coisa que tornaria uma linguagem *diferente* uma escolha melhor. Você vai terminar com uma decisão de stack justificada antes de escrever uma linha 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.
```

Construindo Funcionalidades, Vibe por Vibe



Até agora você já sabe conversar com sua IA e fazer com que ela escreva código. Mas uma funcionalidade de verdade é maior do que um único prompt. A habilidade que separa quem *lança* de quem fica travado é saber como dividir uma funcionalidade em pedaços pequenos e conduzir a IA por eles um de cada vez — mantendo o app funcionando a cada passo.

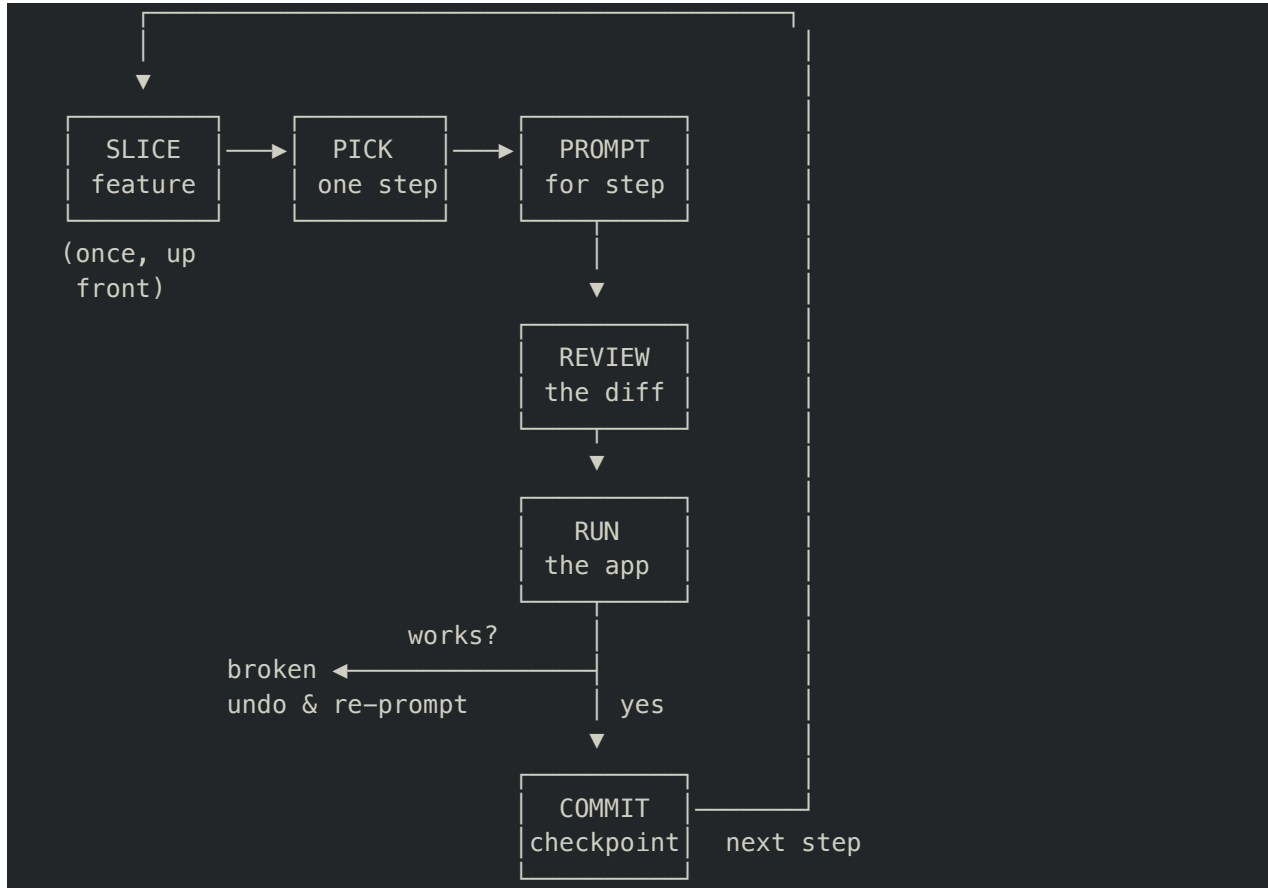
Este capítulo é o loop central de construção. Aprenda uma vez e você vai reutilizá-lo em toda funcionalidade que você construir. É a diferença entre um projeto que cresce de forma constante e um que desmorona sob seu próprio peso no exato momento em que fica interessante.

O Loop de Construção

Aqui está o loop que você vai rodar repetidamente. Ele é pequeno de propósito.

1. **Divida a funcionalidade em pequenos passos verticais.** Cada passo deve ser algo que você consiga construir, rodar e *ver funcionando* em poucos minutos.
2. **Escolha um passo.** Só um. Ignore tudo o mais por enquanto.
3. **Faça o prompt para esse passo.** Seja específico sobre o que muda e o que permanece igual.
4. **Revise a mudança.** Leia o que a IA escreveu. Você não precisa entender cada linha, mas passe os olhos procurando algo obviamente errado.
5. **Rode o app.** Clique de fato no botão, carregue a página, acione o endpoint. Confirme que o passo funciona.
6. **Faça o commit.** Salve um ponto de checagem no controle de versão com uma mensagem curta.
7. **Repita** a partir do passo 2 com o próximo passo — até a funcionalidade estar pronta.

Desenhado como um ciclo, o loop fica assim — você continua girando nele, um pequeno passo por volta, até a funcionalidade estar terminada:



O ponto central é que você nunca fica longe de um app funcionando. Se o passo 4 produz algo quebrado, você só tem uma pequena mudança para desfazer, não uma bagunça emaranhada.

Quem pula os passos 4 a 6 se sente mais rápido por uns vinte minutos. Depois esbarra em um bug que não consegue localizar, porque a última versão *conhecidamente funcional* do app está a cem mudanças de distância. O loop parece lento justamente porque nunca deixa você acumular esse tipo de dívida.

O Que Significa “Fatia Vertical”

Uma **fatia vertical** (vertical slice) é um pedaço fino da funcionalidade que atravessa todo o seu app de ponta a ponta — um pouco de interface, um pouco de lógica, um pouco de dados — em vez de construir uma camada inteira de cada vez.

Abordagem tentadora mas errada (horizontal):

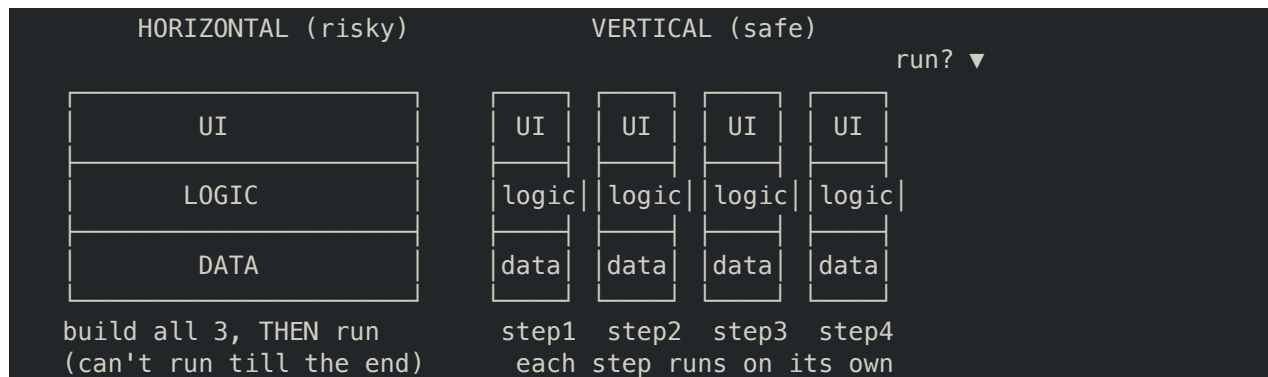
- Construir o esquema de banco de dados inteiro.
- Depois construir toda a lógica de backend.
- Depois construir a interface toda.

Você não consegue rodar *nada* até as três estarem prontas, então não consegue pegar problemas cedo.

Abordagem melhor (vertical):

- Passo 1: um botão que faz a versão mais simples possível da coisa, com dados falsos.
- Passo 2: fazer com que ele use dados reais.
- Passo 3: tratar os casos extremos.

A diferença fica mais fácil de ver lado a lado. A horizontal constrói uma camada inteira de cada vez e não roda nada até o fim; a vertical constrói uma faixa fina e executável atravessando todas as camadas a cada passo:



Cada fatia é executável. É isso que mantém o app sempre funcionando.

Um teste útil para saber se a fatia é boa: **dá para fazer uma demonstração dela?** Se você consegue apontar para a tela e dizer “olha só isso”, é uma fatia vertical de verdade. Se a única demonstração honesta é “bom, o esquema já está no lugar”, você construiu uma camada horizontal e só vai saber se está certo bem mais tarde — geralmente na pior hora possível.

Um Exemplo Trabalhado: Uma Funcionalidade de “Favoritos”

Vamos construir uma pequena funcionalidade do início ao fim: usuários podem marcar itens de uma lista como favoritos e ver uma contagem de quantos eles favoritaram.

Dividindo em fatias

Aqui está como eu dividiria em passos verticais:

1. Adicionar um botão de estrela ao lado de cada item (não faz nada ainda, só aparece).
2. Clicar na estrela alterna sua aparência ligada/desligada (só estado, sem salvar).
3. Mostrar um contador “3 favoritos” que reflete as alternâncias.
4. Salvar os favoritos para que persistam depois de recarregar a página.

Note que cada passo é executável e visível de forma independente. Você poderia parar depois de qualquer um deles e ainda ter um app funcionando.

Passo 1: o botão

Um bom prompt nomeia o arquivo, a mudança e o que *não* deve ser tocado:

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.

Rode o app. Você vê as estrelas? Ótimo. Faça o commit:

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

Passo 2: alternar o 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.

Pense no estado (state) como a memória de curto prazo do app — o que é verdade agora mesmo, como se uma determinada estrela está preenchida no momento. Agora você está lidando com **estado e interface juntos**, o que é normal. (UI, ou interface do usuário, é apenas a parte do app que uma pessoa vê e clica.) A IA provavelmente vai adicionar algo como:

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

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

Clique em algumas estrelas. Elas alternam? Faça o commit.

Passo 3: o 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.

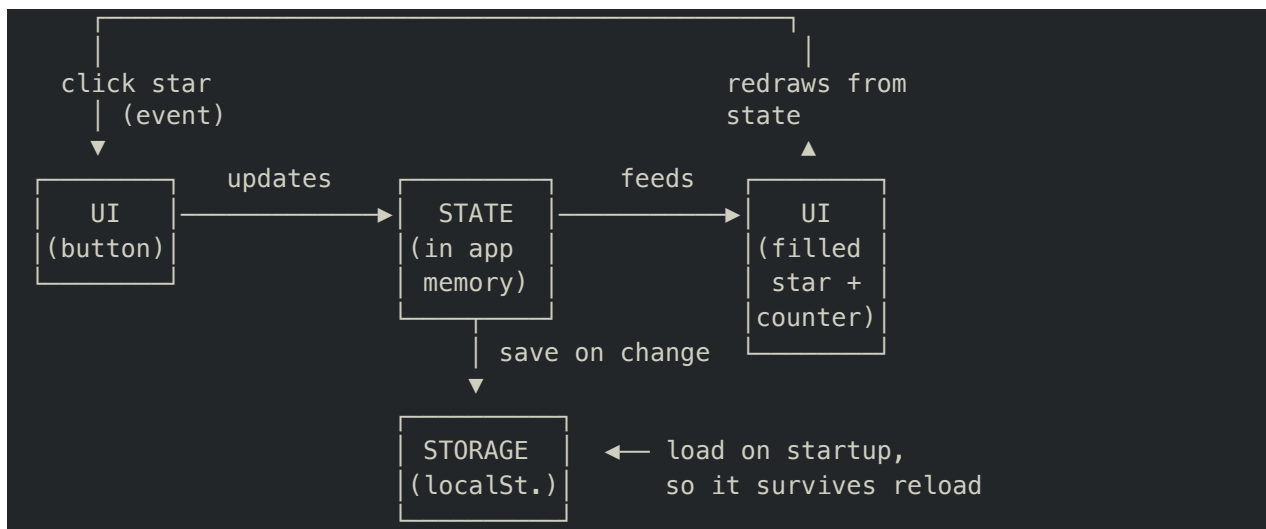
Esse passo pode exigir que a IA **reestruture onde o estado vive** — isso é esperado, e é exatamente por isso que você faz isso como seu próprio pequeno passo. Rode, alterne algumas estrelas, veja o número mudar. Faça o commit.

Passo 4: fazer com 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 é apenas um pequeno caderno que o navegador guarda no próprio computador do visitante, para que o app possa anotar coisas e encontrá-las de novo depois de recarregar a página.) Recarregue a página. Ainda favoritado? Você acabou de lançar uma funcionalidade, uma fatia executável de cada vez.

Aqui está como as peças da funcionalidade concluída conversam entre si. Um clique flui para o estado, a tela é redesenhada a partir do estado, e o estado é espelhado no armazenamento para sobreviver a um recarregamento:



Lendo um Diff Sem Entender Cada Linha

O passo 4 do loop é “revise a mudança”, e é o que as pessoas mais relaxam. Você não precisa ser capaz de escrever o código você mesmo para revisá-lo de forma útil. Você precisa passar os olhos procurando um pequeno conjunto de sinais de alerta.

Quando a IA entrega uma mudança, a forma mais limpa de vê-la é o diff — as linhas exatas adicionadas e removidas:

```
git diff
```

Linhas começando com + foram adicionadas, linhas com – foram removidas. Leia como um revisor, não como um compilador. Você está procurando coisas como:

- **Ela tocou em arquivos que não tinham nada a ver com o pedido?** Você pediu um contador; por que `auth.js` está no diff?
- **Ela apagou algo que parecia importante?** Uma linha removida (–) que você não reconhece merece uma pergunta.
- **Ela mudou silenciosamente algum comportamento que você não pediu?** Novos valores padrão, condições alteradas, “extras úteis”.
- **A forma corresponde ao pedido?** Um pedido de uma frase que produziu duzentas linhas merece uma segunda olhada.

Se algo parecer estranho, você não precisa consertar sozinho — basta perguntar: *“Por que você mudou o `auth.js`? Eu só pedi o contador.”* Metade das vezes a IA vai dizer “boa observação” e desfazer. Esse único hábito pega mais bugs do que qualquer quantidade de tempo encarando código que você não entende completamente.

Quando Deixar a IA Correr Livre vs. Guiar Manualmente

Nem todo passo precisa do mesmo controle no volante.

Deixe a IA correr livre quando:

- O passo é pequeno e bem definido (“adicionar um botão”).
- É um padrão comum que a IA claramente já viu mil vezes.
- Um erro é barato e óbvio (você vai perceber assim que rodar o app).

Guie manualmente — prompts menores, revisão mais próxima — quando:

- A mudança envolve dados que você não consegue recuperar facilmente (apagar registros, migrações).
- A IA fica “consertando” a mesma coisa em círculos.
- O comportamento é sutil e não vai aparecer só de olhar a tela (matemática de valores monetários, permissões, autenticação).
- Você já fez duas tentativas e está *piorando*, não melhorando.

Uma regra confiável: **quanto maior o raio de impacto, menor deve ser o passo.**

Mantenha o App Sempre Funcionando

Essa é a disciplina que torna tudo o mais possível:

- **Rode depois de cada passo.** Não a cada três passos. A cada um.
- **Faça commit só de estados funcionais.** Um commit é um ponto de checagem para o qual você pode voltar. Se um passo quebrar algo que você não consegue consertar em um minuto ou dois, descarte (`git restore` ou reverta) e refaça o prompt com um pedido mais claro — não empilhe um conserto em cima de uma base quebrada.

- **Uma preocupação por passo.** Se você ficar tentado a também “rapidinho consertar aquela outra coisa”, resista. Termine a fatia atual, faça o commit, depois comece um novo passo.
- **Descreva o bug, não faça o diagnóstico.** Quando algo quebra, diga à IA exatamente o que você viu: *“Depois de clicar na segunda estrela, o contador mostra 1, não 2.”* Deixe que ela encontre a causa.

Fazendo Commits em Pequenos Incrementos

Commits pequenos são sua rede de segurança. Cada um deve ser um único passo coerente:

```
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"
```

Se o passo 4 der errado, seus três primeiros passos estão seguros e você quase não perde nada. Commits grandes e raros são a forma como as pessoas acabam com medo de mexer no próprio código.

Quando Uma Fatia É Grande Demais

Às vezes você escolhe “um passo” e ainda assim dá errado — a IA reescreve metade do app, o diff fica ilegível, e ao rodar surgem três bugs novos de uma vez. Isso quase nunca é azar. É um sinal de que a fatia era grande demais.

O conserto é cortar mais fino. Se “adicionar favoritos com persistência” explode, aquilo não era uma fatia; era a funcionalidade inteira disfarçada de um único prompt. Volte para o seu último commit bom e divida:

```
# 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."
```

Uma boa regra prática: se você não consegue descrever o que um passo faz em uma única frase curta, são dois passos. E se o diff de um passo é longo demais para passar os olhos em um minuto, o próximo deveria ser menor. Dividir em fatias é uma habilidade que você aprimora sentindo onde a última fatia ficou grossa demais.

O Ponto Principal

Uma funcionalidade não é um prompt gigante — é uma pilha de fatias pequenas e executáveis. Divida em fatias, faça o prompt para um passo, revise, rode, faça o commit, repita. Deixe a IA correr livre nas coisas fáceis e guie com firmeza perto de qualquer coisa arriscada. Mantenha o app funcionando a cada ponto de checagem, e você sempre estará a um pequeno “desfazer” de distância de um terreno sólido.

No próximo capítulo, vamos ver o que fazer quando um passo não sai como planejado — depuração por vibe.

Recapitulação e Prática

Principais conclusões

- Uma funcionalidade é uma pilha de pequenas fatias verticais, não um prompt gigante — construa, rode e veja cada fatia funcionar antes de seguir em frente.

- Rode o loop de construção sempre: dividir em fatias → escolher um passo → prompt → revisar o diff → rodar o app → fazer commit → repetir.
- Revise diffs como um revisor, não como um compilador — fique atento a arquivos que não deveriam ter sido tocados, mudanças silenciosas de comportamento e um diff muito maior do que o pedido.
- Deixe a IA correr livre em passos pequenos, baratos e comuns; guie manualmente (prompts menores, revisão mais próxima) sempre que o raio de impacto for grande.
- Faça commit só de estados funcionais, em pequenos incrementos — se uma fatia der errado, des-carte e refaça o prompt em vez de empilhar um conserto em cima de uma base quebrada.

Coloque em prática

Escolha uma pequena funcionalidade que você queira em um projeto e divida-a em 3 a 5 passos verticais, escrevendo cada um como uma única frase curta. Construa *apenas o primeiro passo*: faça o prompt para ele, rode `git diff` e passe os olhos na mudança, rode o app para confirmar que funciona, depois faça o commit com uma mensagem de uma linha. Pare por aí por enquanto. O objetivo é sentir uma volta limpa do loop — uma fatia executável, um diff que você realmente leu, um commit para o qual você pode voltar.

```
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.
```

Depurando Com IA



Todo programa que você construir vai quebrar. Isso não é sinal de que você está fazendo algo errado — é o estado normal do software. Bugs não são uma falha de habilidade; eles acontecem quando código real encontra dados reais, usuários reais e casos extremos que ninguém imaginou. A diferença entre uma tarde frustrante e uma correção de cinco minutos é o *método*. Quando você depura fazendo *vibe coding*, você não é quem fica olhando o stack trace até ele fazer sentido — o stack trace é o relatório de erro que seu programa imprime quando trava, e vamos explicar isso melhor mais adiante. Você é o diretor: você reproduz o problema, reúne as evidências certas e entrega para a IA um caso bem definido e enquadrado para que ela faça o trabalho pesado. Este capítulo te dá um processo repetível exatamente para isso — um processo que funciona seja o bug um erro de digitação de uma linha ou uma condição de corrida (race condition) emaranhada três camadas abaixo. (Uma condição de corrida é um bug de tempo: duas coisas acontecem em uma ordem que você não esperava, como duas pessoas disputando o último assento ao mesmo tempo.)

A mentalidade central

O maior erro que iniciantes cometem é colar “não funciona, conserte” no chat e torcer. A IA não consegue ver sua tela, seus dados ou o que você esperava que acontecesse. Ela só sabe o que você conta a ela. Quando você não dá nada a ela, ela faz a única coisa que pode: chuta — e um chute confiante que por acaso está errado é mais perigoso do que nenhuma resposta, porque você vai agir com base nele. Depurar com IA é principalmente sobre **alimentá-la com o contexto certo** para que ela possa raciocinar em vez de chutar.

Um bom relatório de bug — para um humano *ou* uma IA — responde a três perguntas:

- **O que você fez?** (os passos para reproduzir o bug)
- **O que você esperava que acontecesse?**
- **O que realmente aconteceu?** (o erro, a saída errada, a falha)

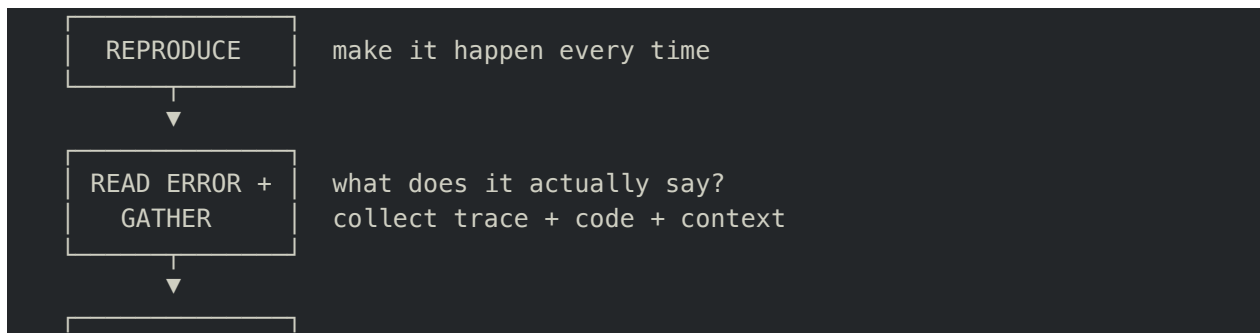
Se você conseguir responder a essas três, já está à frente da maioria das pessoas. O motivo pelo qual essas três coisas importam é que um bug é, por definição, uma *lacuna* entre o que você esperava e o que ocorreu. A IA não consegue encontrar uma lacuna sem enxergar os dois lados dela. “Está quebrado” descreve apenas um lado; “eu esperava uma tela de boas-vindas, recebi uma página em branco e este erro” descreve os dois — e a lacuna praticamente se nomeia sozinha.

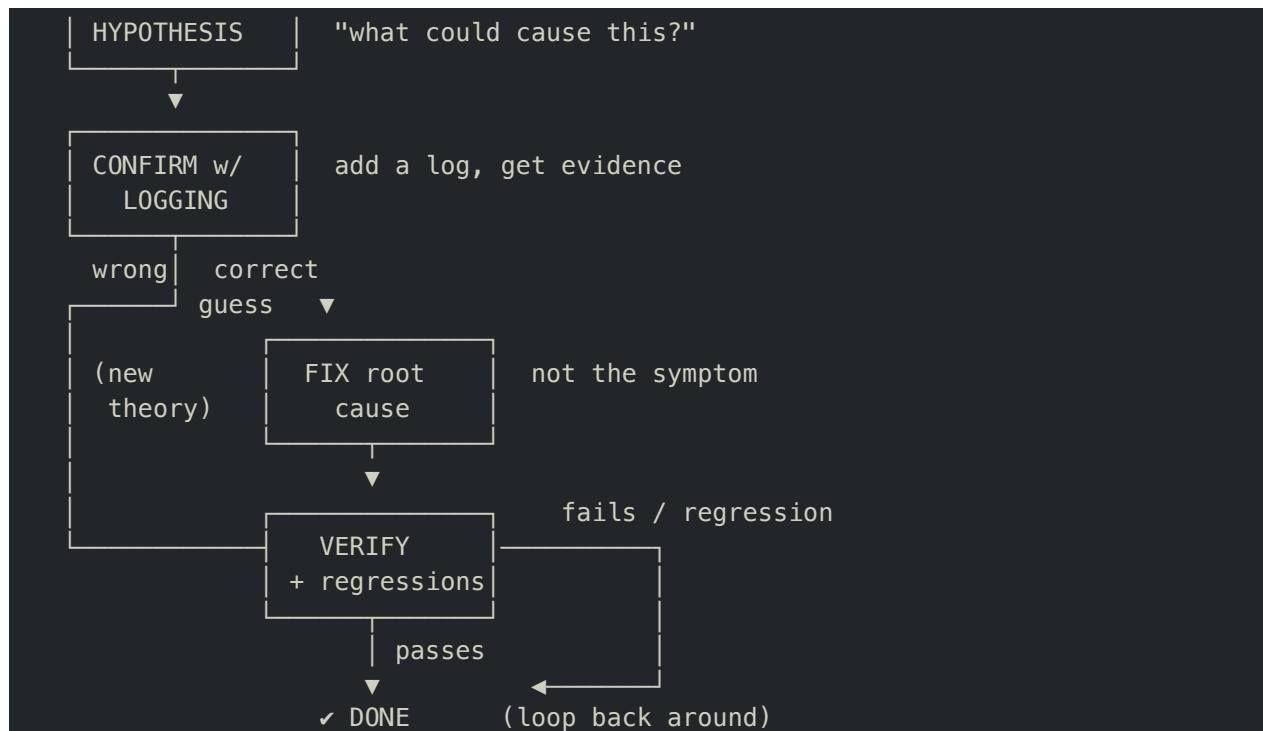
Uma checklist de depuração

Percorra estes passos em ordem. Pular direto para “conserta” é o que faz a IA girar em círculos.

1. **Reproduza de forma confiável.** Encontre os passos exatos que fazem o bug acontecer toda vez. Um bug que você não consegue reproduzir é um bug que você não consegue confirmar que corrigiu — você vai ficar só chutando se ele sumiu ou não. Se ele acontece de forma intermitente, procure a variável que muda o resultado: uma entrada específica, um estado logado versus deslogado, uma rede lenta.
2. **Leia o erro de verdade.** Não passe os olhos por cima. A mensagem de erro e o *topo* do stack trace geralmente indicam o arquivo e a linha onde algo deu errado. Iniciantes costumam rolar a tela passando pela frase mais útil da tela porque ela parece intimidadora.
3. **Reúna o contexto.** Colete o texto do erro, o stack trace, o código relevante e uma descrição de uma linha do que você esperava.
4. **Entregue o quadro completo à IA.** Cole tudo isso numa única mensagem. Deixe que ela forme uma hipótese.
5. **Forme uma hipótese, não uma correção.** Pergunte “o que poderia causar isso?” antes de “conserte isso.” Uma correção errada baseada numa teoria errada desperdiça tempo — e às vezes adiciona um *segundo* bug em cima do primeiro.
6. **Adicione registros (logging) para testar a hipótese.** Confirme o que o código está realmente fazendo antes de mudá-lo.
7. **Faça busca binária no problema.** Reduza *onde* ele quebra cortando repetidamente ao meio a área suspeita.
8. **Corrija a causa raiz, não o sintoma.** Garanta que a correção resolva *por que* quebrou, não apenas o defeito visível.
9. **Verifique a correção e cheque por regressões.** Refaça os passos de reprodução. Depois garanta que funcionalidades próximas ainda funcionam — uma *regressão* é quando uma correção quebra silenciosamente algo que costumava funcionar, como consertar um vazamento e estourar um cano ao lado.

Essa checklist é, na verdade, um único ciclo que você percorre até o bug sumir. Cada etapa entrega evidência para a próxima, e uma verificação que falha só te manda de volta ao ciclo com o que você aprendeu:





A ordem não é arbitrária. Cada etapa produz a evidência de que a próxima precisa. Reproduzir te dá algo para ler; ler te dá contexto para reunir; o contexto dá à IA algo sobre o que formar hipóteses. Pule uma etapa e você estará alimentando a próxima com um buraco.

Alimentando a IA com o contexto certo

É aqui que está a maior parte da sua alavancagem. Ao relatar um bug, inclua:

- **A mensagem de erro, literal.** Copie exatamente — não parafraseie. Parafrasear um erro é como descrever uma foto por telefone: você vai deixar de fora justamente o detalhe que importava.
- **O stack trace**, especialmente as primeiras linhas.
- **O código relevante** — a função que lançou o erro, mais tudo que ela chama.
- **O que você esperava versus o que aconteceu.**

Veja como é, na prática, um prompt de depuração forte:

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:

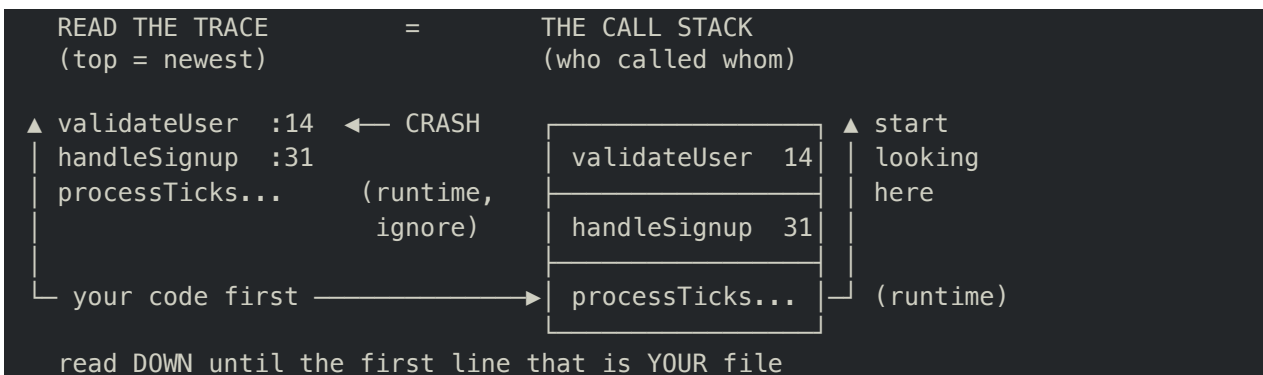

```
[paste the function]
```

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

Perceba a última linha. Você está pedindo uma *hipótese e um passo de confirmação*, não uma edição imediata. Isso mantém a IA honesta.

Ajuda saber como ler o trace que você está colando. Um stack trace é um histórico de chamadas, do mais recente para o mais antigo: a linha do topo é onde o programa de fato explodiu, e cada linha abaixo é a função que chamou a de cima. Então, no exemplo, `validateUser` é onde `email` foi lido de algo indefinido, e `handleSignup` é quem chamou `validateUser`. Os quadros abaixo do seu próprio código — como `processTicksAndRejections` — pertencem ao runtime da linguagem; você geralmente pode ignorá-los. Leia de cima para baixo até encontrar a primeira linha que aponta para o *seu* arquivo, e é quase sempre ali que você deve começar a procurar.

O mesmo trace, lido como uma pilha de chamadas, torna a ordem óbvia. A falha está no topo; cada quadro abaixo é o chamador do quadro acima dele:



Lendo erros que você não entende

Mais cedo ou mais tarde você vai esbarrar num erro que parece sopa de letrinhas — um nome de tipo longo, um stack cheio de coisas internas do framework, uma mensagem num jargão que você nunca viu. Não deixe isso te travar. O erro é dado, não um veredito sobre sua competência. Quando a mensagem em si é opaca, cole-a e peça para a IA *traduzir antes de consertar*:

```
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 o significado em linguagem simples primeiro faz duas coisas. Isso te ensina o padrão, então na próxima vez que você vir esse erro vai reconhecê-lo sozinho — e ao longo de alguns meses é assim que você realmente aprende a ler seu próprio stack. E força a IA a fundamentar a correção no que o erro *realmente diz*, em vez de pular para um remendo genérico. Cuidado com a armadilha de erro mais comum: um erro que aponta para a linha 14 muitas vezes foi *causado* em algum lugar totalmente diferente. A linha no trace é onde o programa percebeu o problema, nem sempre onde o problema começou.

Hipóteses, registros e busca binária

Assim que a IA propuser uma causa, não aceite de cara. **Teste-a de forma barata.** O teste mais barato geralmente é uma linha de log.

Peça à IA para adicionar registros temporários:

“Antes de mudarmos qualquer coisa, adicione um `console.log` (ou `print`) que mostre o valor de `user` logo antes da linha 14, para vermos se ele está realmente indefinido.”

Execute, cole a saída de volta. Agora você tem *evidência*, não uma teoria. Talvez `user` esteja indefinido — confirmando a hipótese. Talvez esteja tudo bem e o verdadeiro culpado esteja um nível acima. De qualquer forma, você avançou. Registre generosamente enquanto investiga: o valor da variável, e um rótulo dizendo qual é ela e onde o log está. Um `console.log(user)` nu que imprime `undefined` é muito menos útil do que `console.log("user at validate.js:14 =", user)`. E lembre-se de remover os logs temporários assim que o bug for corrigido — ruído de depuração deixado para trás é uma bagunça pequena por si só.

Quando o bug vive em algum lugar de um processo longo e você não consegue saber onde, use **busca binária**. Corte a região suspeita ao meio: adicione um log no ponto médio.

A ideia: cada verificação descarta metade do que resta, então um espaço de busca enorme desmorona rápido. Aqui um bug se esconde em algum lugar entre o passo 1 e o passo 8 — três verificações o encurralam:

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

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.
```

O valor parecia correto ali? Então o bug está *depois* dele — pesquise essa metade. Já parecia errado? O bug está *antes* — pesquise essa metade. Cada passo reduz o espaço de busca pela metade, então até um caminho de mil linhas se estreita em cerca de dez verificações. Diga explicitamente à IA: “Ajude-me a bisseccionar isso — qual é o ponto médio onde eu devo colocar um log?” A mesma técnica funciona no histórico, não só no código: se uma funcionalidade funcionava semana passada e quebrou hoje, bisseccione seus commits para encontrar a mudança exata que introduziu o bug.

Causa raiz versus sintoma

A armadilha mais tentadora na depuração é o remendo que faz o erro *desaparecer* sem corrigir a causa. Se `user` está indefinido e você “conserta” isso com `if (user) { ... }`, o erro para — mas agora os cadastros silenciosamente não fazem nada, o que é pior: uma falha barulhenta pelo menos te dizia que algo estava errado, e agora nada diz. A pergunta real é *por que user está indefinido aqui?* Talvez a busca no banco de dados tenha falhado. Talvez um campo tenha sido renomeado. Talvez esteja faltando um `await` mais acima.

Quando a IA propuser uma correção, pergunte a ela:

- “Isso está corrigindo a causa ou escondendo o sintoma?”
- “Qual era o motivo *original* pelo qual o valor estava errado?”
- “Essa mesma causa raiz poderia quebrar outra coisa?”

Uma correção de causa raiz geralmente torna o código *mais simples* ou *mais claro*. Um remendo de sintoma geralmente adiciona uma proteção, um `try/catch` que engole erros, ou um valor padrão que disfarça

o problema real. Desconfie do segundo tipo. Há um sinal que vale a pena aprender: se uma correção adiciona código cujo único trabalho é *evitar* uma falha em vez de *fazer a coisa certa*, você provavelmente está remendando um sintoma. Correções de verdade tendem a eliminar completamente o estado impossível, não apenas sobreviver a ele.

Quando a IA gira em círculos

Às vezes a IA sugere uma correção, ela não funciona, sugere outra, essa também não funciona, e você já está três voltas de “tenta isso em vez daquilo” dentro. Isso é um sinal, não um beco sem saída. Faça isto:

- **Pare e reinicie o contexto.** Comece uma mensagem nova resumindo tudo o que você agora *sabe* com certeza (confirmado por logs), não tudo o que você já tentou. Um chat longo cheio de tentativas fracassadas envenena ativamente a próxima resposta da IA — ela fica fazendo correspondência de padrões com os becos sem saída no próprio histórico dela.
- **Declare o que você já descartou.** “Confirmamos que user está indefinido e a consulta ao banco de dados retorna null. A string da consulta parece correta. Então o problema está a montante da consulta.” Isso impede que a IA volte a sugerir becos sem saída.
- **Peça para ela explicar, não consertar.** “Me explique, passo a passo, como user recebe seu valor desde a requisição até a linha 14.” Forçar uma explicação muitas vezes revela a lacuna.
- **Adicione mais registros, não mais chutes.** Quando as teorias acabam, evidência é o caminho a seguir.
- **Encolha o problema.** Peça à IA para te ajudar a construir o menor trecho possível que ainda reproduza o bug. Muitas vezes o bug fica óbvio assim que o ruído desaparece — e se ele não se reproduzir na versão pequena, você aprendeu que a causa está no que você removeu.

Entendendo a correção antes de mantê-la

Uma correção que funciona mas que você não entende é um empréstimo, não uma solução — e os juros vencem na próxima vez que algo próximo quebrar. Como a IA faz a digitação, é fácil aceitar um sinal verde e seguir em frente sem nunca saber *por que* o código agora está verde. Resista a isso. Antes de manter uma correção, faça a IA prestar contas dela:

- **“Explique em uma ou duas frases por que isso corrige o problema.”** Se a explicação for vaga — “isso deixa mais robusto”, “isso trata o caso extremo” — questione. Uma explicação de verdade nomeia a causa específica e como a mudança a elimina.
- **“Qual era o bug, em termos de causa e efeito?”** Você deve conseguir repetir a resposta com suas próprias palavras. Se não conseguir, você ainda não entende seu próprio programa.
- **“O que mais toca nesse trecho de código que eu deveria testar de novo?”** Isso transforma uma correção em uma verificação de regressão.

Isso é a filosofia de entender o que você entrega aplicada ao seu caso mais difícil. Você não precisa ser capaz de *escrever* a correção do zero — esse é o trabalho da IA. Mas você precisa segurar a história de causa e efeito na cabeça, porque é você quem vai estar lá às 2 da manhã quando quebrar de novo, e a única coisa que ajuda nessa hora é saber como a coisa realmente funciona.

O ponto principal

Depurar com IA não é sobre saber a resposta. É sobre executar um processo disciplinado: reproduzir, reunir contexto, formular hipótese, confirmar com registros, bisseccionar e corrigir a causa raiz — e depois entender a correção o suficiente para mantê-la honestamente. Seu trabalho como diretor é manter a investigação apoiada em evidências e reconhecer quando o ciclo ficou estagnado. Faça isso, e a maioria dos bugs deixa de ser mistério e passa a ser apenas mais um passo na construção de software real.

Recapitulação e Prática

Principais conclusões

- Um bug é uma lacuna entre o que você esperava e o que aconteceu — dê à IA os dois lados mais os passos exatos, nunca apenas “está quebrado.”
- Cole o erro de forma literal e o topo do stack trace; a primeira linha que aponta para o *seu* arquivo é quase sempre onde começar.
- Confirme antes de mudar: adicione uma linha de log rotulada para testar uma hipótese, e bisseccione (no código ou no histórico de commits) para reduzir o espaço de busca pela metade a cada passo.
- Corrija a causa raiz, não o sintoma — uma proteção ou try/catch que só impede uma falha geralmente disfarça o problema real.
- Quando a IA entrar em loop, reinicie o contexto para o que você *sabe* com certeza, peça para ela explicar em vez de consertar, e adicione evidência em vez de chutes.

Experimente

Na próxima vez que algo quebrar, escreva o relatório do bug *antes* de pedir uma correção. Num chat novo, dê à IA três coisas: os passos para reproduzir, o que você esperava e o que realmente aconteceu — com o erro copiado literalmente e os primeiros quadros do stack incluídos. Depois, em vez de pedir “conserta”, peça uma hipótese e uma forma barata de confirmá-la (geralmente uma linha de log). Execute essa verificação, cole a saída de volta, e só então deixe a IA propor uma correção. Perceba o quanto o caminho fica mais direto do que “não funciona, me ajuda.”

```
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?
```

Infraestrutura de servidor



Você fez vibe coding de algo que funciona no seu laptop. Agora você quer que outras pessoas usem. Essa lacuna — entre “roda na minha máquina” e “roda na internet” — é do que trata a infraestrutura. A boa notícia: você não precisa de uma carreira em DevOps para atravessá-la. Você precisa de um modelo mental funcional e da disposição de deixar a IA cuidar da configuração detalhada.

Este capítulo desmistifica o mundo da hospedagem, apresenta os trade-offs (as trocas — o que você ganha e o que perde em cada escolha) em linguagem simples e entrega um caminho padrão que leva um construtor solo do protótipo à produção sem afogar você em opções.

Os cinco lugares onde o código pode viver

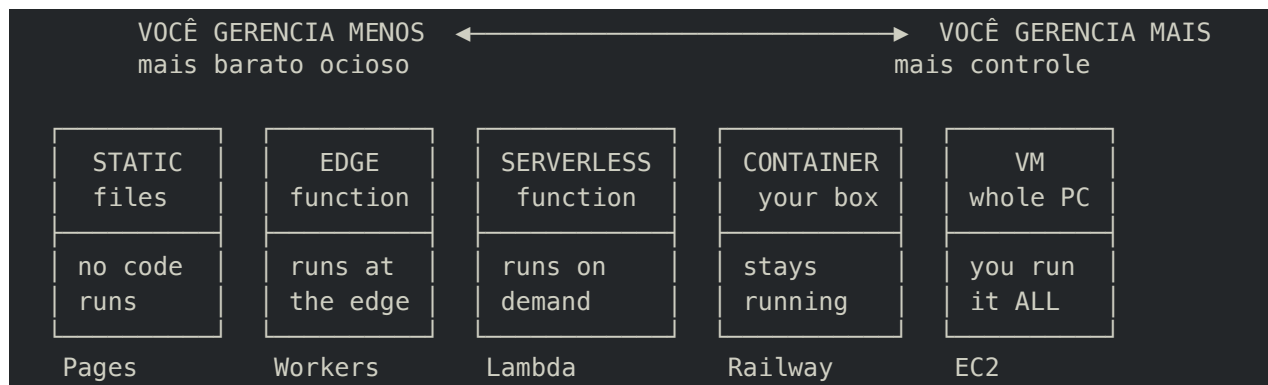
Quase todo produto de hospedagem na Terra é uma variação de uma dessas cinco categorias. Aprenda-as e as páginas de marketing param de ser confusas.

- **Hospedagem estática** — Arquivos simples (HTML (HyperText Markup Language, a estrutura da página), CSS (Cascading Style Sheets, a estilização), JavaScript, imagens) entregues diretamente ao navegador. Nenhum servidor “roda” seu código; ele apenas entrega arquivos. Pense nisso como uma pasta na internet. Como não há nada para travar nem nada para corrigir, é a coisa mais confiável que você pode colocar online. Exemplos: GitHub Pages, Cloudflare Pages, Netlify, Vercel (modo estático).
- **Funções de borda (edge functions)** — Pequenos trechos de código que rodam em servidores fisicamente próximos ao seu usuário, espalhados pelo mundo todo. Eles iniciam quase instantaneamente e são ótimos para lógica leve: verificações de autenticação, redirecionamentos, pequenas chama-

das de API (Application Programming Interface — a porta de entrada que um programa usa para conversar com outro). A pegadinha: eles rodam em um ambiente enxuto, então nem toda biblioteca funciona ali. Exemplos: Cloudflare Workers, Vercel Edge Functions, Deno Deploy.

- **Funções serverless (sem servidor)** — Código que roda sob demanda e desaparece quando está ocioso. Você é cobrado por requisição, não por hora. Você escreve uma função; a plataforma cuida das máquinas. Ótimo até uma única requisição precisar rodar por minutos — a maioria das plataformas corta a função após um tempo limite (geralmente entre 10 e 60 segundos). Exemplos: AWS Lambda, Google Cloud Functions, Vercel Functions.
- **Contêineres (containers)** — Seu app junto com todo o seu ambiente (a versão certa da linguagem, bibliotecas, ferramentas de sistema) empacotado em uma caixa portátil chamada contêiner. A plataforma mantém essa caixa rodando. Mais controle, um pouco mais de configuração. O arquivo que define essa caixa é um Dockerfile, e a IA escreve isso muito bem. Exemplos: Fly.io, Railway, Render, Google Cloud Run.
- **Máquinas virtuais (VMs)** — Um computador inteiro que você aluga em um data center. Você instala e gerencia tudo sozinho — o sistema operacional, as atualizações de segurança, o servidor web, tudo. Controle máximo, responsabilidade máxima. Exemplos: AWS EC2, DigitalOcean Droplets, Hetzner.

Lado a lado, as cinco categorias trocam *menos para gerenciar* por *mais controle* à medida que você desce:

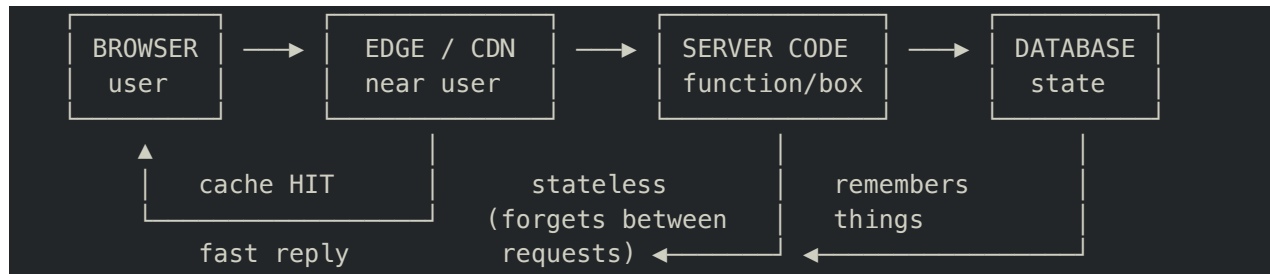


Uma regra prática para custo: quanto mais a plataforma gerencia por você, mais barato é quando ninguém está usando seu app, e mais ela se preocupa quando todo mundo está.

Alguns termos técnicos, decodificados

- **Cold start (partida a frio)** — Quando o código não roda há um tempo, a plataforma precisa acordá-lo. Essa primeira requisição é mais lenta (de uma fração de segundo a alguns segundos). Funções de borda quase não têm isso; funções serverless às vezes têm; contêineres e VMs sempre ativos não têm. Para a maioria dos apps isso é invisível; para um fluxo de checkout ágil, pode importar.
- **Escalonamento (scaling)** — Lidar com mais usuários sem cair. Escalonamento *automático* significa que a plataforma adiciona capacidade por você. Escalonamento *manual* significa que você clica em botões ou o app simplesmente quebra sob carga. “Escala a zero” é uma frase relacionada que você vai ver bastante: quando ninguém está usando seu app, ele não custa nada.
- **Estado (state)** — Dados que precisam persistir (um banco de dados, arquivos enviados, uma sessão de usuário). As camadas estática e serverless costumam ser “sem estado” (stateless) — elas esquecem tudo entre requisições — então o estado vive em um banco de dados ou serviço de armazenamento separado. É por isso que quase todo app real é “sua hospedagem mais um banco de dados em algum lugar”, não uma caixa mágica única.
- **Região (region)** — Em que parte do mundo seu código e seus dados ficam fisicamente. Isso afeta a velocidade (mais perto é mais rápido) e às vezes a lei (alguns dados precisam permanecer em um país específico). Para um primeiro projeto, escolha a região mais próxima dos seus usuários e siga em frente.

Aqui está toda a jornada de uma requisição, e onde cada parte vive. O navegador conversa com uma borda próxima, que entrega arquivos em cache ou acorda seu código, que recorre ao banco de dados para tudo o que precisa lembrar:



A tabela de trade-offs

Tipo de hospedagem	Custo (ocioso □ ocupado)	Escalonamento	Cold starts	Complexidade	Melhor para
Estática	Grátis □ muito barato	Automático, sem esforço	Nenhum	Mínima	Sites institucionais, documentação, blogs, frontends
Funções de borda	Camada grátis □ barato	Automático, global	Insignificante	Baixa	Autenticação, redirecionamentos, APIs leves, personalização
Serverless	\$0 ocioso □ escala com o uso	Automático	Às vezes perceptível	Baixa-média	APIs, webhooks, tráfego imprevisível e em picos
Contêineres	Paga enquanto roda	Automático (relativamente), configurável	Nenhum (se mantido ativo)	Média	Backends completos, tarefas de longa duração, runtimes customizados
VMs	Paga 24/7	Manual	Nenhum	Máxima	Hardware especial, software legado, controle total

O padrão: à medida que você desce na tabela, você troca *simplicidade e baixo custo ocioso* por *controle e flexibilidade*. A maioria dos vibe coders deve lutar para permanecer perto do topo.

Quando escolher qual

- **Só um site ou frontend?** Hospedagem estática. Pronto. É grátis, rápida e quase impossível de quebrar.

- **Precisa de um pouco de lógica de backend — um formulário de contato, uma chave de API mantida em segredo, uma verificação de autenticação?** Funções de borda ou serverless. Você fica em uma camada grátis generosa e escala a zero quando ninguém está usando.
- **Um app de verdade com banco de dados, tarefas em segundo plano, ou um framework que espera um servidor rodando continuamente?** Uma plataforma de contêineres como Railway, Render ou Fly.io. Esse é o ponto ideal para a maioria dos projetos que são “produtos de verdade”.
- **Você precisa de uma GPU específica, software de sistema especial, ou controle total?** Uma VM. Só vá para lá quando algo acima genuinamente não conseguir fazer o trabalho, porque agora você é o administrador de sistema.

Uma regra útil: **escolha a opção mais simples que consiga fazer o trabalho, e só desça um nível quando esbarrar em uma parede real** — não uma imaginada.

Como é, de fato, uma “parede real”? Alguns exemplos honestos: seu build produz um servidor que precisa continuar rodando (um backend de chat mantendo conexões ao vivo), então o estático não serve. Ou uma função fica batendo no limite de tempo porque uma requisição faz trabalho pesado de verdade (redimensionar um vídeo, gerar um PDF), então você sobe para um contêiner. Ou o limite de requisições da sua camada grátis está genuinamente esgotado por tráfego real. Note o que essas situações têm em comum: cada uma é uma medição, não um palpite. “Pode ficar grande algum dia” não é uma parede.

Como a IA monta deploys e configuração

É aqui que o vibe coding brilha. Configuração de deploy é exatamente o tipo de trabalho preciso, bem documentado e repetitivo em que a IA é excelente. Você direciona; ela escreve.

Prompts eficazes soam assim:

- “Adicione um Dockerfile para este app Node e uma configuração do Railway para fazer o deploy.”
- “Escreva um workflow do GitHub Actions que faça o deploy da pasta dist/ no Cloudflare Pages a cada push na branch main.”
- “Estou recebendo um timeout de cold-start na minha função serverless — o que provavelmente está errado e como corrijo a configuração?”

Aqui está o tipo de arquivo que a IA vai produzir para você — um workflow de deploy que publica um site estático automaticamente sempre que você faz push de 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
```

Você não precisa memorizar essa sintaxe. Você precisa reconhecer o que ela faz: a cada push na main,

ela baixa seu código, faz o build e publica o resultado. Quando algo quebra, você cola o erro de volta para a IA e pede que ela corrija a configuração.

Muitos construtores pulam o arquivo do GitHub Actions completamente no início e simplesmente fazem o deploy do próprio terminal com um comando. Essa costuma ser a forma mais rápida de ver algo no ar no primeiro dia. O mesmo deploy do Cloudflare Pages fica assim pela linha de comando:

```
# instale a CLI (Command Line Interface, interface de linha de comando) uma vez
npm install -g wrangler

# faça o build do seu site, depois envie a pasta de saída ao ar
npm run build
wrangler pages deploy dist --project-name=my-app
```

A primeira execução vai pedir que você faça login pelo navegador; depois disso é um único comando. O trade-off é simples: o comando de terminal é ótimo para “publicar agora”, e o arquivo do GitHub Actions é ótimo para “publicar automaticamente para sempre”. A maioria dos projetos começa com o primeiro e adiciona o segundo quando os deploys viram rotina. Peça à IA o que fizer mais sentido para o momento em que você está.

Algumas dicas de direcionamento que evitam dor de cabeça real:

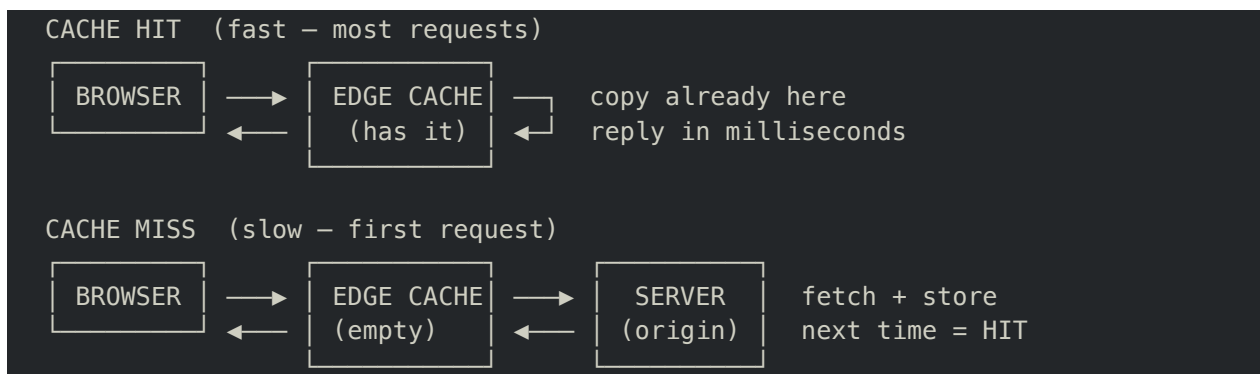
- **Faça a IA explicar antes de agir.** “Antes de escrever a configuração, me diga qual tipo de hospedagem combina com este projeto e por quê.” Isso pega excesso de engenharia cedo.
- **Mantenha segredos fora do código.** Chaves de API e senhas vão nas variáveis de ambiente / cofre de segredos da plataforma, nunca em arquivos que você commita. Peça à IA para usar uma referência de segredo, como no `CF_API_TOKEN` acima.
- **Faça deploy cedo, faça deploy sempre.** Coloque um “hello world” no ar no primeiro dia. Um pipeline que já funciona desde o início é muito mais fácil de depurar do que um montado só no lançamento.
- **Pergunte de antemão qual é a história do rollback (reversão).** “Se esse deploy quebrar, como eu volto para a versão que funcionava?” Boas plataformas mantêm seus deploys anteriores a um clique de distância. Saber que o botão de desfazer existe antes de precisar dele transforma um pânico às 2 da manhã em um dar de ombros.

Um caminho padrão recomendado para o construtor solo

Aqui está uma rota que funcionou para inúmeros projetos de uma pessoa só. Cada etapa é dada apenas quando você realmente precisa dela.

1. **Protótipo: estático + um serviço de backend gerenciado.** Coloque seu frontend em hospedagem estática (Cloudflare Pages, Vercel ou Netlify). Para dados e autenticação, apoie-se em um backend gerenciado como Supabase ou Firebase para pular a tarefa de rodar um banco de dados você mesmo. Custo: praticamente zero.
2. **Produção inicial: adicione funções de borda ou serverless.** Quando você precisar de lógica no lado do servidor — chamadas de API secretas, webhooks, autenticação customizada — adicione funções na mesma plataforma. Ainda escala a zero, ainda barato, ainda um único deploy.
3. **Produto de verdade: mova o backend para uma plataforma de contêineres.** Quando o app crescer além das funções (processos de longa duração, um framework que quer um servidor persistente, tarefas em segundo plano), containerize-o e faça o deploy no Railway, Render ou Fly.io. Peça à IA para escrever o `Dockerfile` e a configuração da plataforma.
4. **Escalar ou especializar: só então considere VMs ou um provedor de nuvem como a AWS.** É aqui que a complexidade (e sua fatura) dá um salto. A maioria dos construtores solo nunca precisa dessa etapa, e isso é sinal de estar fazendo as coisas certo — não uma limitação.

Uma CDN (Content Delivery Network, rede de distribuição de conteúdo) fica na frente desse caminho e muda a velocidade drasticamente dependendo de já ter uma cópia ou não. Um cache HIT (acerto de cache) responde instantaneamente pela borda; um cache MISS (erro de cache) precisa viajar de volta até seu servidor primeiro:



O erro a evitar é começar pela etapa 4 porque é o que supostamente “engenheiros de verdade” usam. O oposto é verdade: começar simples é o movimento de alguém experiente. Você sempre pode avançar para mais controle, mas não recupera facilmente os meses que perderia lutando contra uma infraestrutura de que não precisava.

O que “gerenciado” realmente significa (e o que ainda é seu)

Um serviço gerenciado é aquele em que a plataforma cuida das partes chatas e arriscadas por você — os servidores, as correções de segurança, o escalonamento, os backups. “Serverless”, “banco de dados gerenciado” e “plataforma como serviço” são todos sabores do mesmo acordo: você entrega as operações e continua construindo. Para um construtor solo, isso não é preguiça; é alavancagem. As horas que você não gasta aplicando atualizações de segurança do Linux são horas que você gasta no produto de verdade.

Mas “gerenciado” não é “mágica”. Algumas coisas continuam sendo sua responsabilidade, não importa quanto a plataforma cuide:

- **Seus dados.** A plataforma mantém o banco de dados rodando; garantir que você tem um backup que você realmente testou restaurar é com você. Clique no botão “exportar” uma vez e confirme que o arquivo não está vazio.
- **Seus segredos.** As chaves de API vivem no cofre de segredos da plataforma, mas rotacionar uma que vazou, e nunca colar uma em um repositório público ou em uma captura de tela, é sua tarefa.
- **Sua fatura.** O escalonamento automático é maravilhoso até escalar automaticamente sua conta. Configure alertas e limites de gasto — a próxima seção é inteiramente sobre isso, porque é o tipo de falha que mais dói.
- **Seu entendimento.** Quando algo quebra à meia-noite, “a plataforma cuida disso” só vai até certo ponto. Você não precisa ser um especialista, mas deveria conseguir ler os logs e descrever a falha para a IA com clareza. Essa é toda a filosofia de “entenda o que você entrega” em uma frase.

A mentalidade saudável: terceirize as *operações*, não o *entendimento*. Deixe a plataforma rodar as máquinas; você continua sendo a pessoa que sabe o que as máquinas estão fazendo.

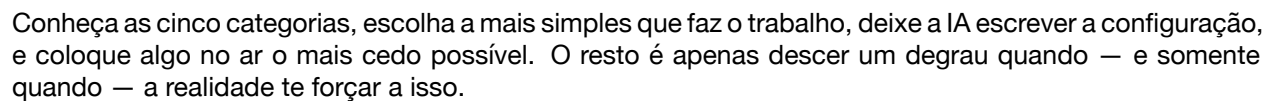
Uma checklist rápida de pré-lançamento

Antes de dizer para as pessoas visitarem sua URL ao vivo, percorra esta lista. Nada dela exige experiência em DevOps — a maior parte é um prompt para a IA ou um clique em um painel.

- **Realmente carrega.** Abra a URL (Uniform Resource Locator, o endereço da web) real em um navegador limpo (ou no seu celular), não apenas na sua pré-visualização local. O número de lançamentos “funciona na minha máquina” que dão 404 em público é lendário.

- Isso não é uma auditoria completa de produção — é a lista curta que pega os erros vergonhosos, caros e fáceis de evitar. Rode-a a cada lançamento até virar um reflexo.

Infraestrutura é uma escada, não um salto. Estático no topo, VMs embaixo, e a maior parte do seu trabalho vivendo feliz nos degraus superiores.



Principais conclusões

- 91

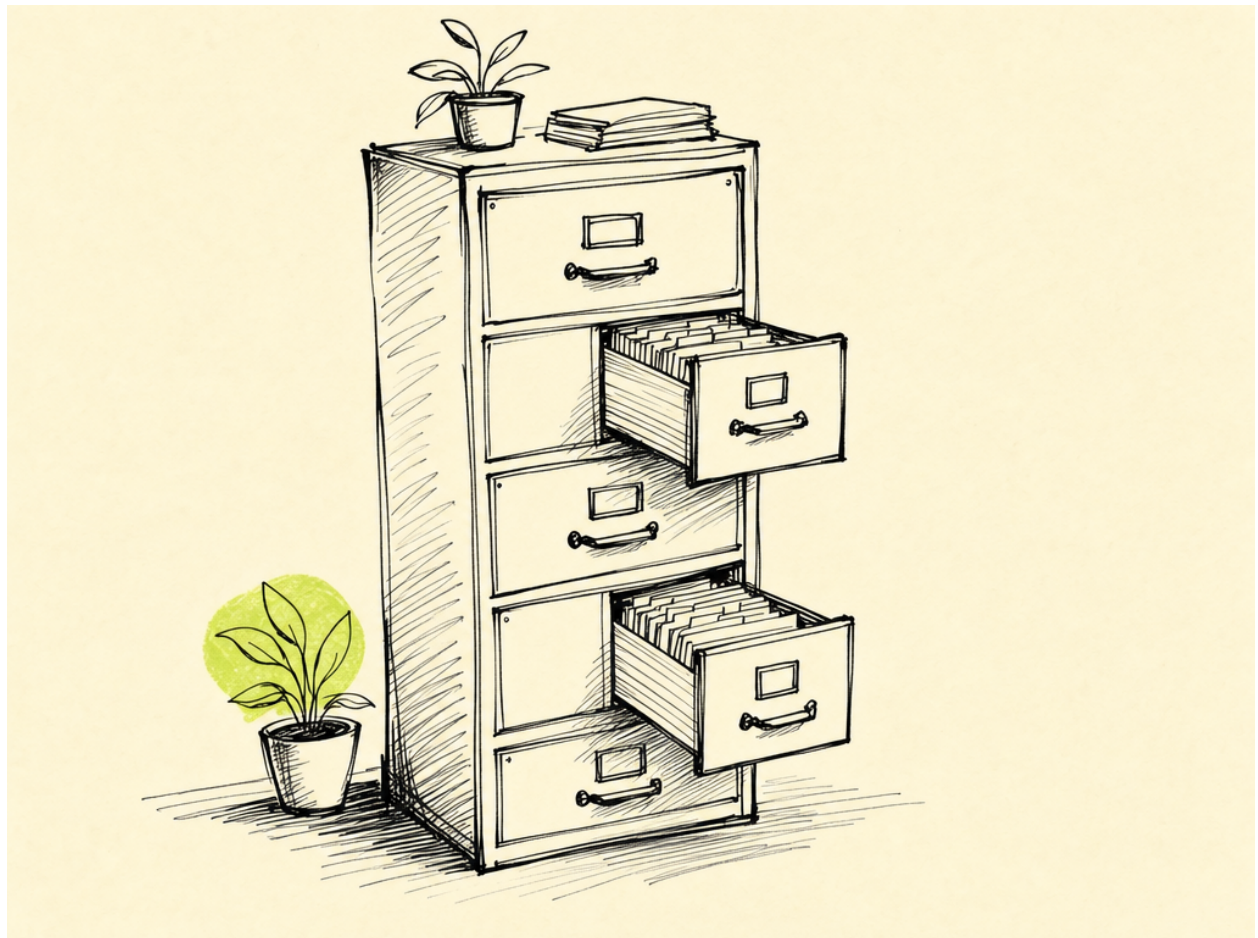
- Limite o gasto, confirme o HTTPS e saiba como reverter *antes* de lançar.

Coloque em prática

Pegue um projeto pequeno que você já tem (mesmo uma única página HTML) e faça o deploy dele em um host estático. Depois peça à IA para gerar um workflow do GitHub Actions que faça o redeploy a cada push, leia o YAML gerado linha por linha, e confirme que consegue encontrar o botão “voltar ao deploy anterior” no painel do seu 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.
```

Infraestrutura de Banco de Dados



Todo aplicativo de verdade acaba precisando lembrar de coisas — usuários, pedidos, mensagens, configurações. Essa memória mora em um banco de dados. Para quem está começando, “banco de dados” pode parecer uma caixa-preta guardada por especialistas. Não é. Este capítulo desmistifica a camada de dados para que você faça boas escolhas e direcione a IA a fazer o trabalho pesado com segurança.

A coisa a internalizar cedo: código é substituível, dados geralmente não são. Você pode reescrever uma função com bug em uma tarde, mas uma coluna que você corrompeu na terça-feira passada — ou um backup que nunca fez — pode ter ido embora para sempre. Essa assimetria é o motivo pelo qual a camada

de dados recompensa um pouco de entendimento prévio mais do que quase qualquer outra parte da sua stack.

As Três Formas de Armazenamento de Dados

A maioria dos bancos de dados se encaixa em três famílias. Você não precisa memorizar produtos — precisa reconhecer qual *forma* se encaixa no seu problema.

Tipo	Exemplos	Melhor para	Trade-offs
Relacional (SQL)	PostgreSQL, MySQL, SQLite	Dados estruturados com relacionamentos; qualquer coisa crítica em termos de dinheiro ou precisão	Você precisa desenhar um schema antecipadamente; escalar escritas muito grandes é mais difícil
Documento / NoSQL	MongoDB, Firestore, DynamoDB	Dados flexíveis ou aninhados, iteração rápida, formatos de registro variados	Fácil de criar dados inconsistentes; relacionamentos e relatórios ficam complicados
Chave-Valor	Redis, Memcached, Cloudflare KV	Cache, sessões, contadores, buscas rápidas por uma chave conhecida	Sem busca por conteúdo; geralmente não é sua fonte de verdade

As três formas armazenam dados de maneiras genuinamente diferentes. Imagine-as lado a lado:

RELACIONAL (SQL)

linhas + colunas

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

tabelas referenciam uma à outra

DOCUMENTO (NoSQL)

blobs autocontidos

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

tudo aninhado

CHAVE-VALOR

uma chave → um valor

key	value
session	"x9f2"
count	"42"

sem busca interna – procure só por chave

Pense em um banco de dados relacional como um conjunto de planilhas que sabem como se referenciar mutuamente. A linguagem que você usa para conversar com um deles é SQL (Structured Query Language, ou Linguagem de Consulta Estruturada). Um bom padrão para 90% dos projetos: **comece com um banco de dados relacional (Postgres)**. Ele é maduro, previsível, impõe estrutura e lida com os relacionamentos que apps de verdade inevitavelmente desenvolvem. Recorra a bancos de documentos quando seus dados forem genuinamente sem forma fixa, e a bancos chave-valor como uma camada de *apoio* (cache), não seu armazenamento principal.

A armadilha a evitar é escolher pela vibe. “NoSQL (bancos de dados que não usam tabelas SQL — eles armazenam registros flexíveis, de formato livre) escala melhor” só é verdade para um conjunto estreito de problemas que a maioria dos apps nunca encontra, e mesmo assim o preço é abrir mão das garantias de consistência que tornam os dados confiáveis. Escolha a forma que combina com seus *dados*, não a que soa mais impressionante em um vídeo de design de sistemas. Quando estiver em dúvida, relacional é a aposta segura — você sempre pode adicionar um cache ou um banco de documentos depois, mas desamaranhar um modelo de dados NoSQL confuso é doloroso.

Gerenciado vs Auto-hospedado

Você pode rodar um banco de dados você mesmo em um servidor, ou alugar um **gerenciado** (Supabase, Neon, PlanetScale, RDS, Cloudflare D1) onde o provedor cuida de backups, patches e disponibilidade.

Para vibe coding, **comece sempre gerenciado**. Auto-hospedar um banco de dados significa que agora você é responsável por backups, atualizações de segurança, espaço em disco e recuperação às 3 da manhã. Isso é trabalho operacional de verdade que distrai da construção. Serviços gerenciados custam um pouco de dinheiro e economizam enorme tempo e risco. Só migre para auto-hospedagem quando tiver um motivo concreto e as habilidades para sustentá-lo.

Quando você escolhe um provedor gerenciado, vale a pena saber pelo que está realmente pagando, porque as páginas de marketing se confundem. Algumas coisas realmente diferem:

- **Modelo de conexão.** Plataformas serverless (Cloudflare Workers, funções Vercel) abrem e fecham conexões de banco de dados constantemente, o que pode esgotar um pool de conexões clássico do Postgres. Provedores como Neon e Supabase oferecem um *pooler de conexões* (PgBouncer ou equivalente) exatamente para isso — use a string de conexão via pool em código serverless.
- **Branching e backups.** Neon e PlanetScale permitem “ramificar” o banco de dados como no Git, para que você possa testar uma migração em uma cópia. Confirme se a recuperação point-in-time está ativa, e saiba até quando ela alcança.
- **Aprisionamento (lock-in).** D1 e DynamoDB são excelentes, mas prendem você a uma única nuvem. Postgres puro (Supabase, Neon, RDS) é portátil — seu schema e suas queries se movem para qualquer host Postgres. Para a maioria dos iniciantes, portátil é a aposta mais segura no longo prazo.

Fundamentos de Design de Schema

Um *schema* é a planta baixa dos seus dados: quais tabelas existem, quais colunas elas têm e como se relacionam. Um bom design de schema é principalmente senso comum mais alguns hábitos.

- **Uma tabela por “coisa.”** Usuários, posts, pedidos — cada um recebe sua própria tabela.
- **Toda linha precisa de um ID único** (uma chave primária).
- **Conecte tabelas com chaves estrangeiras**, não copiando dados de um lado para o outro.
- **Escolha tipos honestos.** Dinheiro é decimal, não float. Datas são timestamps, não texto.
- **Não duplique.** Se o email de um usuário mora em cinco tabelas, eventualmente você terá cinco emails diferentes.

Aqui está um schema simples para um blog, do tipo que a IA vai gerar para você:

```
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);
```

Aquela linha `author_id ... REFERENCES users(id)` é a chave estrangeira. Ela conecta cada post de volta a exatamente um usuário real, então o próprio banco de dados se recusa a armazenar um post sem um autor válido:



Repare em `REFERENCES` (uma chave estrangeira ligando um post a um usuário real) e nas restrições `NOT NULL` e `UNIQUE`. Esses são guardrails que o banco de dados aplica por você, para que dados ruins não consigam entrar mesmo quando seu código tem um bug.

Uma restrição que vale a pena adicionar deliberadamente é o que acontece quando o “pai” desaparece. Por padrão, apagar um usuário cujos posts ainda o referenciam vai falhar — o que geralmente é o que você quer. Mas você pode ser explícito: `REFERENCES users(id) ON DELETE CASCADE` apaga os posts também, enquanto `ON DELETE RESTRICT` bloqueia a exclusão. Escolher de propósito é melhor do que descobrir o comportamento padrão do jeito difícil em produção. O mesmo instinto de “deixe o banco de dados te proteger” se aplica a uma restrição `CHECK` (`CHECK (price >= 0)`) — ela recusa dados sem sentido na origem, em vez de confiar que todo caminho de código vai validar.

Migrações: Mudando o Schema com Segurança

Seu schema vai mudar — você vai adicionar uma coluna, renomear uma tabela, adicionar um índice. Pense nisso como edições rastreadas de um documento compartilhado: toda mudança é registrada em ordem, para que a cópia de todo mundo possa ser atualizada da mesma forma. Uma **migração** é um script versionado e repetível que descreve uma dessas mudanças. Migrações importam porque o schema no seu laptop, no laptop do seu colega de equipe e em produção precisam permanecer idênticos.

As regras que mantêm migrações seguras:

- **Nunca edite o banco de dados manualmente em produção.** Escreva uma migração.
- **Migrações avançam.** Cada uma é um arquivo novo; você não reescreve as antigas que já rodaram.
- **Faça mudanças destrutivas com cuidado.** Descartar uma coluna ou tabela apaga dados permanentemente. Tenha um backup primeiro.
- **Mudanças aditivas são as mais seguras.** Adicionar uma coluna anulável raramente quebra alguma coisa; renomear ou remover uma pode quebrar código em execução.

Migrações formam uma cadeia ordenada: cada uma é um passo numerado que move cada cópia do banco de dados de um estado conhecido para o próximo, para que seu laptop, o de um colega de equipe e a produção acabem todos idênticos:



Ferramentas como Prisma, Drizzle ou migrações do Rails geram e rastreiam isso para você. Deixe que elas façam.

Uma migração tem duas metades: a mudança a aplicar (up) e como desfazê-la (down). Uma ferramenta confiável gera as duas, para que um deploy ruim possa voltar atrás de forma limpa:

```
-- up: adiciona uma bio opcional, seguro para aplicar ao vivo
ALTER TABLE users ADD COLUMN bio TEXT;

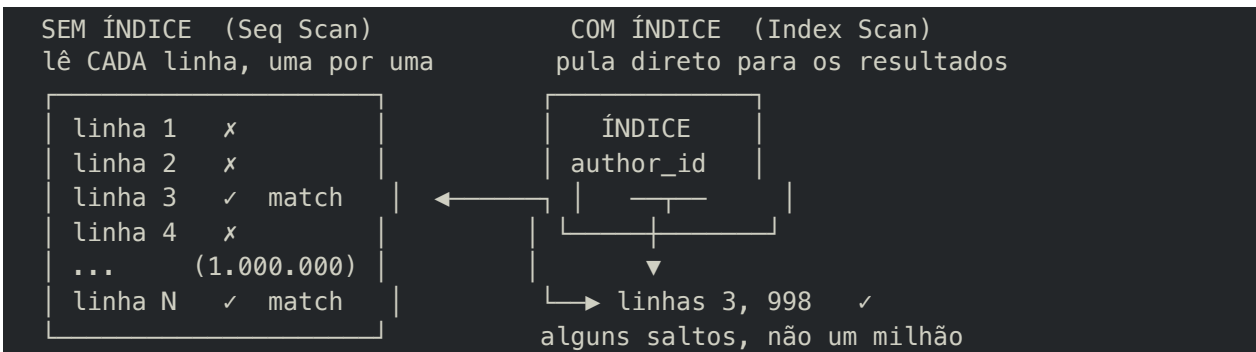
-- down: reverte
ALTER TABLE users DROP COLUMN bio;
```

As migrações genuinamente perigosas são *renomeações* e *mudanças de tipo*, porque o código antigo e o novo schema discordam brevemente. O padrão profissional é dividi-las em passos aditivos: adicione a nova coluna, preencha os dados retroativamente, mude o código para lê-la, e só descarte a coluna antiga em um deploy *posterior*, uma vez que nada mais a referencie. Parece lento, mas é a diferença entre um lançamento tranquilo e um pico de erros de página 500 para cada usuário no meio do deploy.

Indexação

Um **índice** é como o índice remissivo no final de um livro: ele deixa o banco de dados pular direto para as linhas correspondentes em vez de varrer todas elas. Sem índices, as consultas ficam mais lentas conforme os dados crescem — tudo bem com 100 linhas, doloroso com 1.000.000.

Sem um índice, o banco de dados checka cada linha em ordem; com um índice, ele pula direto para o resultado, da mesma forma que o índice de um livro te manda para a página certa em vez de te fazer ler do início ao fim:



A orientação prática:

- Indexe colunas que você frequentemente usa para **filtrar** ou fazer **join** (ex.: `author_id`, `email`).
- Chaves primárias são indexadas automaticamente.
- **Não indexe tudo** — cada índice acelera leituras, mas desacelera escritas e usa espaço.
- Adicione índices quando você perceber uma consulta lenta, não preventivamente para toda coluna.

A maneira honesta de descobrir se um índice ajuda é perguntar ao banco de dados, não adivinhar. Todo motor SQL tem EXPLAIN (no Postgres, EXPLAIN ANALYZE), que mostra o plano que ele vai usar:

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

Se a saída disser Seq Scan em uma tabela grande, o banco de dados está lendo cada linha — um sinal de que um índice ajudaria. Depois de adicionar um, deveria mudar para um Index Scan. Esta é uma ótima tarefa para delegar à IA: cole a consulta lenta e a saída do EXPLAIN, e pergunte qual índice adicionar e por quê. Você mantém o controle entendendo a resposta, não memorizando teoria de índices.

Cache

Um **cache** armazena a resposta para uma pergunta cara, para que você não precise perguntar de novo. Se sua página inicial roda a mesma consulta pesada para cada visitante, guarde o resultado em cache no Redis por 60 segundos e sirva-o instantaneamente.

Cache é uma otimização de performance, não uma fonte de verdade. A armadilha clássica de iniciante é esquecer a **invalidação de cache** — a cópia em cache fica desatualizada e os usuários veem dados antigos. Regra geral: faça cache só quando tiver medido um gargalo real, e sempre tenha uma regra clara para quando o valor em cache expira ou é atualizado.

Há duas estratégias limpas, e misturá-las causa a maioria dos bugs de cache. **Expiração baseada em tempo (TTL — Time To Live, como uma data de validade carimbada no valor em cache)** diz “este valor é válido por 60 segundos, depois descarte” — simples, e bom quando dados um pouco desatualizados são inofensivos (uma contagem de visualizações, um feed da página inicial). **Invalidação baseada em eventos** diz “apague o valor em cache no momento em que os dados subjacentes mudarem” — correto, mas você precisa se lembrar de fazer isso em *todo* caminho de escrita, que é exatamente onde as pessoas escorregam. Na dúvida, prefira um TTL curto: um valor que está desatualizado por no máximo um minuto é um bug menor do que um valor desatualizado para sempre porque você esqueceu uma invalidação.

Filas

Uma **fila** permite dizer “faça isso depois” em vez de “faça isso agora.” Quando um usuário se cadastra, você não quer que ele fique esperando enquanto você envia um email de boas-vindas e redimensiona seu avatar. Coloque essas tarefas em uma fila (RabbitMQ, BullMQ, Cloudflare Queues, SQS); um worker em segundo plano as pega. O usuário recebe uma resposta instantânea.

Use uma fila para qualquer tarefa que seja lenta, possa falhar e ser reprocessada, ou não precise terminar antes de você responder ao usuário.

O problema que a maioria dos iniciantes encontra é que um job enfileirado pode rodar *mais de uma vez* — uma falha de rede faz o worker tentar de novo, e agora o email de boas-vindas é enviado duas vezes. A solução é tornar os jobs **idempotentes**: desenhe-os para que rodar o mesmo job duas vezes tenha o mesmo efeito de rodá-lo uma vez (verifique “eu já enviei isso?” antes de enviar). Também decida de antemão o que acontece com um job que continua falhando; filas maduras roteiam isso para uma *dead-letter queue* (fila de mensagens mortas), para que uma mensagem envenenada não bloqueie tudo atrás dela.

Erros Comuns de Iniciante

- **Recorrer ao NoSQL porque soa moderno.** A maioria dos apps tem relacionamentos; SQL lida melhor com eles.
- **Armazenar dinheiro como ponto flutuante.** $0.1 + 0.2$ famosamente não é 0.3 . Use tipos decimais.
- **Nenhum backup.** Serviços gerenciados geralmente fazem backup automaticamente — confirme isso, não assuma.
- **Colocar segredos ou arquivos enormes no banco de dados.** Arquivos pertencem a armazenamento de objetos (S3, R2); o banco de dados armazena o *link*.
- **Deixar a IA escrever consultas cruas com entrada do usuário colada diretamente.** Isso causa injeção de SQL. Use consultas parametrizadas.
- **A consulta N+1.** Um loop que roda uma consulta por item (buscar 100 posts, depois 100 buscas separadas de autores) sobrecarrega o banco de dados. Busque dados relacionados em uma única consulta com um join, ou agrupe as buscas.
- **Nunca fazer teste de carga.** Uma consulta que é instantânea com as 50 linhas do seu laptop pode rastejar com os 5 milhões da produção. Popule uma quantidade realista de dados falsos antes de confiar na performance.

Direcionando a IA para Construir Sua Camada de Dados

A IA é excelente em design de schema e escrita de consultas — se você der a ela as restrições contra as quais projetar. Não diga “faça um banco de dados.” Descreva o domínio, os relacionamentos e o que não é negociável.

Um prompt forte se parece com isto:

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.

Depois **revise a saída** com a checklist deste capítulo: tipos honestos, chaves estrangeiras e restrições presentes, índices sensatos, nenhum dado duplicado. Ao pedir consultas, exija explicitamente **consultas parametrizadas** para que a entrada do usuário nunca seja concatenada em SQL. E, para qualquer migração que descarte ou renomeie algo, peça para a IA sinalizar isso, para que você possa fazer backup primeiro.

O banco de dados é a parte do seu app onde os erros são mais difíceis de desfazer — dados, uma vez corrompidos ou perdidos, nem sempre voltam. É exatamente por isso que esses fundamentos valem a pena conhecer: não para que você escreva cada linha sozinho, mas para que você possa direcionar a IA com confiança e capturar os movimentos perigosos antes que eles vão para produção.

Recapitulação e Prática

Principais lições

- A maioria dos apps precisa de apenas uma das três formas de armazenamento — relacional, documento ou chave-valor — e relacional é a escolha padrão segura.
- Comece com um banco de dados gerenciado; você mantém os dados e as decisões de schema, a plataforma mantém os servidores funcionando.
- Um bom design de schema significa tipos honestos, chaves estrangeiras e restrições, índices sensatos e nenhum dado duplicado.
- Trate toda migração como uma porta de mão única: faça backup antes de qualquer coisa que descarte ou renomeie, e revise de perto migrações escritas por IA.
- Sempre exija consultas parametrizadas para que a entrada do usuário nunca seja concatenada em SQL.

Coloque em prática

Escolha uma ideia de app pequena e escreva primeiro suas entidades e relacionamentos em português simples. Depois entregue essa descrição para a IA e peça um schema mais uma migração de avanço — e revise o resultado contra a checklist deste capítulo antes de rodar qualquer coisa.

Design a relational schema for a personal bookmarks app.

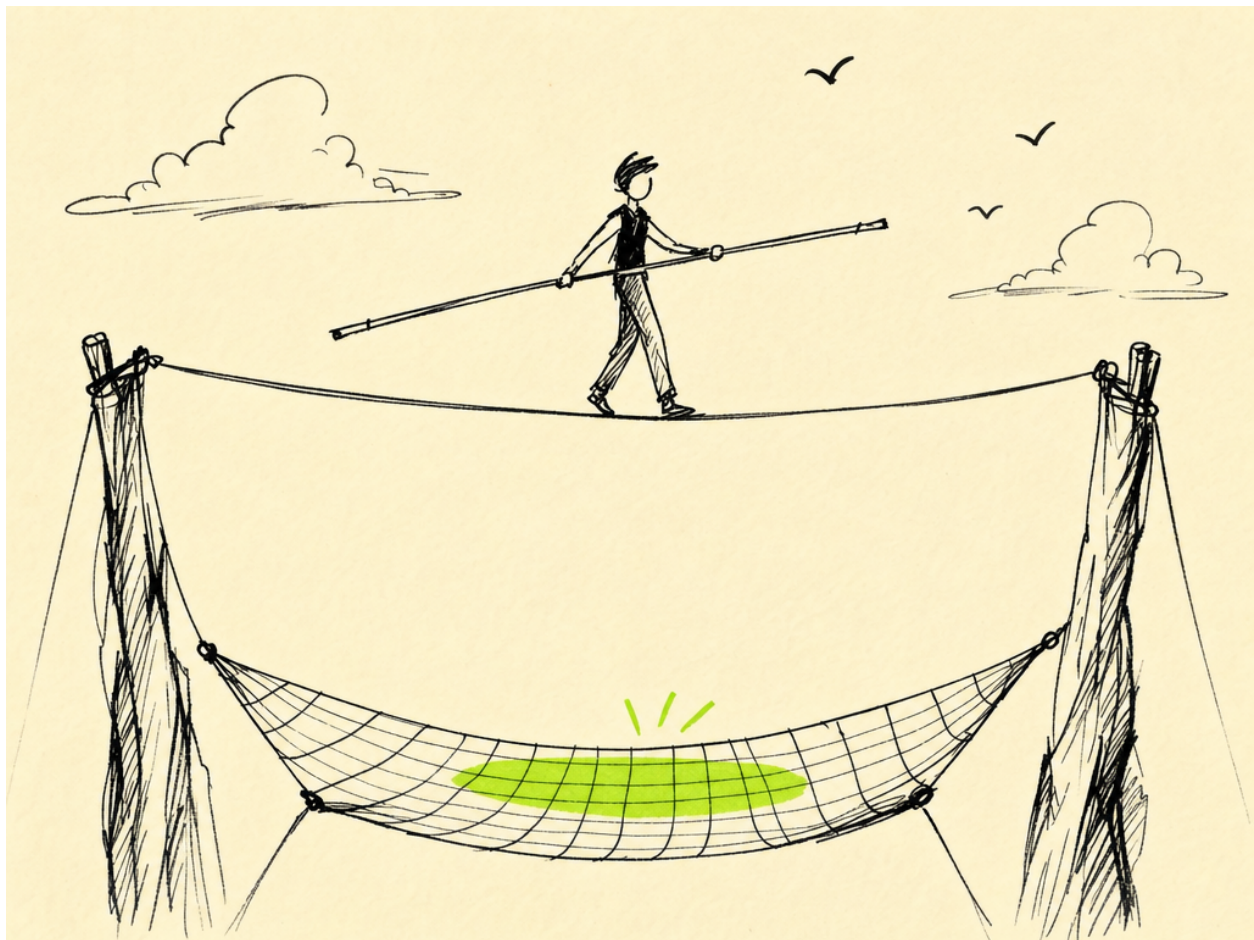
Entities and relationships:

- 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.

Guardrails, Testes e Revisões



O vibe coding permite que você avance rápido. O perigo é que você também pode avançar rápido na direção errada — e como a IA soa confiante, talvez você só perceba quando algo quebrar na frente de um usuário real. Guardrails são a forma de manter a velocidade sem os acidentes.

Este capítulo não é sobre transformar seu projeto em uma burocracia. É sobre um punhado de verificações baratas e automáticas que capturam os erros da IA (e os seus) antes que cheguem à produção. Configure-as uma vez, e elas trabalham em segundo plano para sempre. Toda a filosofia deste capítulo cabe em uma linha: **nunca lance algo que você não entende**. Guardrails são o que tornam “entender” barato o suficiente para que você realmente faça isso sempre.

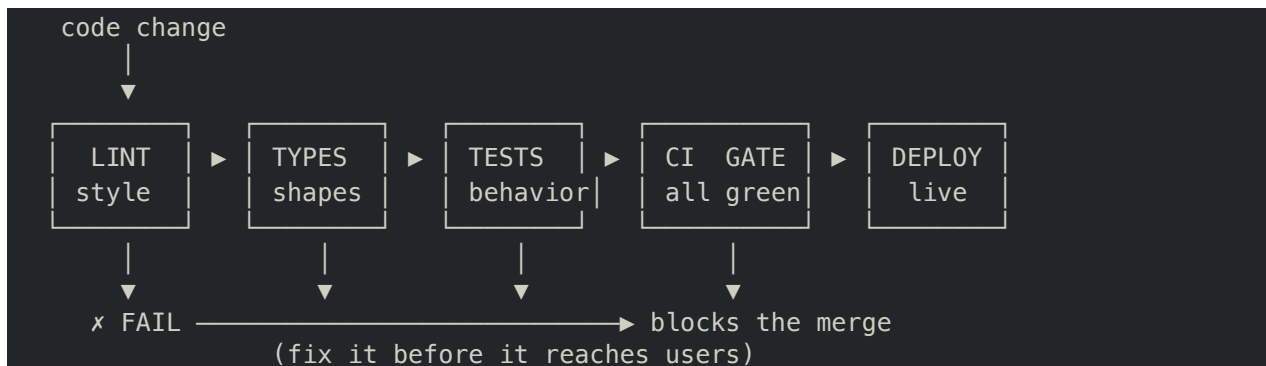
Por que você precisa de guardrails ao dirigir a IA

A IA não sabe o que seu aplicativo *deveria* fazer. Ela faz correspondência de padrões com o que parece plausível. Isso significa que ela vai, alegremente:

- Apagar uma verificação que não entende “para corrigir o erro”
- Refatorar três arquivos quando você pediu para mudar um
- Introduzir um bug sutil que o caminho feliz não revela
- Dizer com confiança que terminou quando nunca rodou o código

Guardrails transformam “confie em mim” em “prove”. O objetivo é um sistema em que seja *difícil* quebrar as coisas por acidente, para que você possa continuar dizendo “só tenta aí” sem prender a respiração.

Pense nos guardrails como um pipeline pelo qual toda mudança precisa passar antes de chegar aos usuários reais. Cada portão é barato e automático; qualquer um deles pode barrar uma mudança ruim na hora:

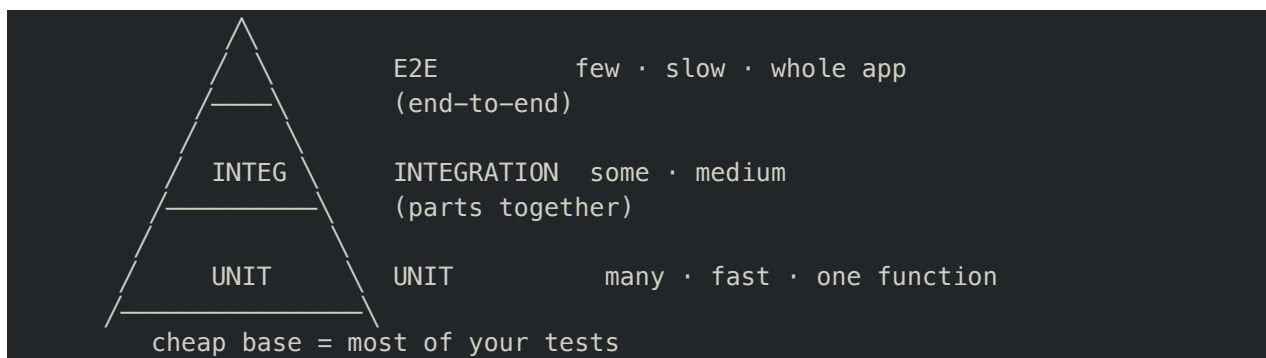


Há uma razão mais profunda pela qual isso importa mais com IA do que com um colega humano. Um humano que apaga uma verificação de validação geralmente sabe que está correndo um risco, e sente o peso disso. A IA não sente nada. Ela vai remover uma proteção, reescrever um arquivo e relatar sucesso no mesmo tom animado, seja a mudança brilhante ou catastrófica. Você não consegue ler sua linguagem corporal, porque ela não tem nenhuma. Os guardrails são a linguagem corporal dela — são o único sinal honesto que você tem sobre se o trabalho é de fato sólido.

Testes: sua rede de segurança

Um teste é código que verifica se outro código faz o que você espera. Uma vez escrito, ele roda em segundos e te avisa imediatamente se algo regrediu.

Uma suíte de testes saudável se parece com uma pirâmide: muitos testes unitários pequenos e rápidos na base, menos testes de integração no meio, e apenas um punhado de testes lentos de ponta a ponta no topo. A maior parte da sua segurança vem barata, da base:



Você não precisa de 100% de cobertura. Comece testando as coisas que mais doeriam se quebrassem: login, pagamentos, dados que não podem ser regenerados, a lógica de negócio central. A IA é excelente em escrever testes — muitas vezes melhor do que em escrever funcionalidades, porque o comportamento esperado é concreto.

Aqui está um prompt que funciona bem:

```
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.
```

Essa última linha importa. Sempre peça para a IA *rodar* os testes, não apenas escrevê-los. Um teste que nunca foi executado é um palpite. Também vale a pena ser específico sobre *quais* casos cobrir, porque, deixada por conta própria, a IA tende a escrever três variações do caminho feliz e chamar isso de completo. Os casos que de fato pegam bugs são os chatos: entrada vazia, o valor limite, a coisa que deveria lançar um erro.

Um exemplo simples do que ela deveria produzir:

```
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();  
  });  
});
```

Quando você mudar uma funcionalidade mais tarde, rode os testes primeiro. Se eles ficarem vermelhos, você encontrou a quebra antes dos seus usuários. Quando a IA corrige um bug, peça para ela adicionar um teste que falhe com o comportamento antigo — assim o bug não pode voltar sorrateiramente. Esse hábito tem um nome: teste de regressão. Cada bug que você encontra é uma lição pela qual o código já pagou; um teste de regressão é como você mantém a lição.

Um aviso: um teste também pode estar errado. Se a IA escrever um teste que afirma que o comportamento *com bug* está correto, ele vai passar para sempre enquanto seu aplicativo continua quebrado. Então leia as asserções, não apenas os checks verdes. Um teste que passa só significa “o código faz o que o teste diz” — você ainda precisa confirmar que o teste diz a coisa certa.

Verificação de tipos e linting: erros pegos de graça

Duas ferramentas capturam categorias inteiras de erros com quase nenhum esforço da sua parte:

- **Verificação de tipos** (TypeScript, mypy, etc.) captura “você passou uma string onde era esperado um número” e “esse valor pode ser undefined”. A IA adora esquecer casos extremos que os tipos tornam impossíveis.
- **Linting** (ESLint, Ruff, etc.) captura variáveis não usadas, código inalcançável e desvio de estilo. Isso mantém um código tocado por IA de virar lentamente uma bagunça.

Configure essas ferramentas cedo e diga à IA para obedecê-las: *“Run the type checker and the linter, and fix anything they report before you call this done.”* Trate uma verificação de tipos limpa como parte da definição de “terminado”.

Fique atento às saídas fáceis e preguiçosas, no entanto. Quando a verificação de tipos reclama, a correção genuinamente certa é lidar com o caso que ela encontrou — o valor undefined, o tipo errado. A correção rápida é silenciar a reclamação com `// @ts-ignore`, `any`, ou um comentário `# type: ignore`, e a IA recorre a isso constantemente porque faz o texto vermelho desaparecer. Isso também torna a verificação inútil exatamente onde ela estava tentando ajudar. Se você ver um comentário de supressão aparecer em um diff, trate-o como uma pergunta, não como uma solução: *“Why is this needed? Fix the underlying type instead of ignoring it.”*

Limitando o escopo das mudanças da IA

A maior causa isolada de dano induzido por IA é deixá-la mudar coisas demais de uma vez. Um diff de 400 linhas espalhado por nove arquivos é impossível de revisar e fácil de esconder um bug.

Mantenha as mudanças pequenas e contidas:

- Peça **uma mudança lógica de cada vez**, não “já que você está aí, também...”
- Diga explicitamente: *“Only touch src/auth.js. Don’t refactor anything else.”*
- Faça commit de estados funcionais com frequência, para sempre ter um ponto limpo para o qual voltar.
- Quando ela propuser uma grande reescrita, peça para dividir o trabalho em etapas que você possa revisar uma por uma.

Um escopo pequeno torna o próximo passo — ler o diff — de fato viável. Há um efeito cumulativo aqui: commits pequenos também têm um raio de explosão pequeno. Se uma mudança bem delimitada acaba estando errada, você reverte um commit e perde dez minutos. Se uma mudança extensa está errada, ou você gasta uma hora desembaraçando o que quebrou, ou joga fora uma tarde de trabalho. Disciplina de escopo é um seguro barato que você compra antes de saber que vai precisar.

Lendo diffs com espírito crítico

Um diff é a lista exata do que mudou: linhas vermelhas removidas, linhas verdes adicionadas. Ler diffs é o hábito de maior alavancagem no vibe coding, porque é onde você pega a IA fazendo algo que você não pediu.

Você não precisa entender cada linha. Procure por estes sinais de alerta:

- **Arquivos que você não esperava que mudassem.** Por que um arquivo de configuração foi tocado?
- **Código apagado.** Ela removeu uma verificação de validação, um tratador de erro, ou um teste “para fazer passar”?
- **Segredos ou valores fixos no código.** Chaves de API (Application Programming Interface — Interface de Programação de Aplicações), senhas, ou URLs (Uniform Resource Locators — endereços web) que não deveriam estar no código.
- **Verificações desativadas.** Um teste pulado, uma proteção comentada, um `// @ts-ignore`, ou uma permissão afrouxada.
- **Aumento de escopo.** Mudanças distantes do que você pediu.

Se algo parecer estranho, pergunte: *“Explain why you changed X — I didn’t ask for that.”* Faça a IA justificar exclusões especialmente. “Estava causando um erro” é uma razão para entender o erro, não para apagar a verificação.

Uma forma prática de criar o hábito: leia o diff *antes* de aceitar a mudança, nunca depois. Assim que o código é mesclado e o aplicativo parece funcionar, você nunca mais vai voltar e lê-lo — não há atrito forçando isso. O momento da revisão precisa ser o momento da decisão. Ferramentas ajudam aqui: `git diff --stat` te dá um resumo de uma linha por arquivo, para que um arquivo inesperado salte aos olhos imediatamente, e revisar em uma página de pull request mostra a mudança inteira em uma única visão rolável, em vez de espalhada pelo seu editor.

Portões de revisão: uma checklist antes de mesclar

Um portão de revisão é uma lista curta de coisas que precisam ser verdade antes de uma mudança ir ao ar. Passe por ela você mesmo, ou peça para a IA confirmar cada item:

- ☐ Eu li o diff e entendo o que mudou e por quê
- ☐ A mudança está limitada ao que eu de fato pedi
- ☐ Os testes passam (e o novo comportamento tem um teste)
- ☐ A verificação de tipos e o linter estão limpos
- ☐ Nenhum segredo, log de depuração ou código comentado ficou para trás
- ☐ Nenhuma verificação de segurança foi removida silenciosamente
- ☐ Eu testei a mudança eu mesmo, pelo menos o caminho feliz

Se você não consegue marcar uma caixa, você não terminou.

A checklist é deliberadamente curta porque um portão que você pula não é portão nenhum. A tentação, especialmente quando você está avançando rápido e a mudança “parece boa”, é deixar passar sem checar. Resista a isso em qualquer coisa voltada ao usuário. Um truque útil é fazer a IA preencher isso para você — cole a lista e pergunte: *“Go through this review gate for the change you just made. For each item, say PASS or FAIL with one line of evidence.”* Ela não consegue escrever honestamente “os testes passam” sem tê-los rodado, e forçá-la a citar evidências linha a linha transforma um vago “parece bom” em algo que você pode auditar.

Fundamentos de CI: torne as verificações automáticas

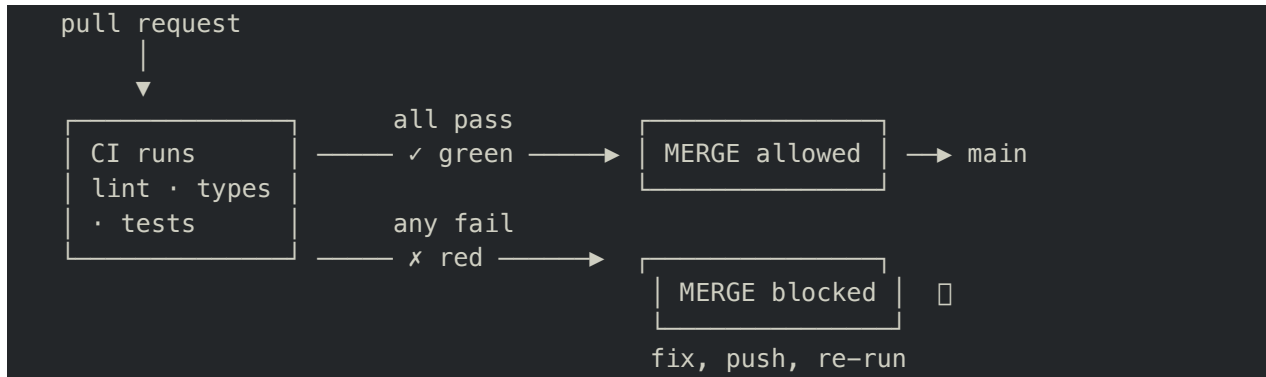
CI (Continuous Integration — Integração Contínua) é um robô que roda suas verificações toda vez que você envia código, para que você não possa *esquecer* de rodá-las. A maioria das plataformas resume isso a um único arquivo. Aqui está um exemplo mínimo do GitHub Actions:

```
# .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
```

Agora todo push é submetido a lint, verificação de tipos e testes automaticamente. Se algo falhar, você vê uma marca vermelha antes que o código possa ser mesclado. Você pode até pedir para a IA escrever esse arquivo para a sua stack — “Add a GitHub Actions workflow that runs our lint, type check, and tests on every pull request.”

Com a proteção de branch ativada, uma verificação vermelha não apenas te avisa — ela trava fisicamente o botão de merge até que a build volte a ficar verde:



O ponto do CI não é a automação por si só — é que ela tira as verificações da sua memória e as coloca no pipeline, onde não podem ser puladas em uma noite cansada. Duas melhorias valem a pena assim que o básico estiver rodando verde. Primeiro, transforme as verificações em um status *obrigatório* na sua branch principal (uma “regra de proteção de branch” no GitHub), para que uma build vermelha literalmente bloqueie o botão de merge em vez de apenas mostrar um aviso que você pode ignorar. Segundo, quando o CI falhar, cole o log da falha direto para a IA: “CI failed on this run, here’s the output — diagnose and fix it.” O log é um relatório de bug preciso, gerado por máquina, que é exatamente o tipo de entrada em que a IA é melhor para agir.

Hábitos seguros por padrão

Junte tudo em reflexos que te permitem avançar rápido sem medo:

- **Trabalhe em uma branch, não diretamente em produção.** Branches são de graça; produção quebrada não é.
- **Faça commits pequenos e funcionais com frequência.** Commits frequentes tornam o rollback trivial.
- **Sempre peça para a IA rodar o que escreveu** — os testes, o aplicativo, o verificador de tipos.
- **Mantenha um terminal para a IA e outro para você**, para poder verificar de forma independente.
- **Nunca deixe uma mudança ir ao ar sem ter sido lida por você.** A velocidade vem de bons guardrails, não de pular a revisão.

Guardrails não te deixam mais lento — eles são o que permite você dizer “sim, manda ver” sem um nó no estômago. Configure-os uma vez, e a IA pode correr enquanto a rede de segurança segura as quedas.

Quando um guardrail grita: faça a triagem antes de corrigir

Guardrails só são úteis se você reage bem a eles, e o modo de falha é previsível: a build fica vermelha, você entra em pânico um pouco, e cola o erro para a IA com “conserta isso”. Às vezes isso funciona. Muitas vezes produz uma correção que faz o texto vermelho sumir sem resolver o que estava realmente errado — a IA enfraquece o teste, amplia um tipo, ou envolve a chamada que falha em um try/catch que engole o problema.

Antes de deixá-la “consertar” qualquer coisa, gaste dez segundos em triagem. Na prática, existem apenas três tipos de falha de guardrail:

- **Um bug real que você acabou de introduzir.** A verificação está fazendo o seu trabalho. Corrija o código, não a verificação.
- **Um teste que agora está desatualizado.** Você mudou o comportamento de propósito, e a assertão antiga não corresponde mais. Atualize o teste para afirmar o *novo* comportamento correto — deliberadamente, não por reflexo.
- **Uma verificação instável ou errada.** Genuinamente raro. Desconfie de si mesmo aqui; esse é o balde onde as pessoas jogam bugs reais porque é o que permite seguir em frente.

A instrução que mantém a IA honesta é fazê-la diagnosticar antes de mudar qualquer coisa:

```
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.
```

Essa pausa é a diferença entre um guardrail que te protege e um guardrail que você treinou silenciosamente a IA a driblar. Uma verificação vermelha é informação; a pior coisa que você pode fazer é pagar para deixá-la verde sem ler o que ela estava te dizendo.

Recapitulação e prática

Principais conclusões

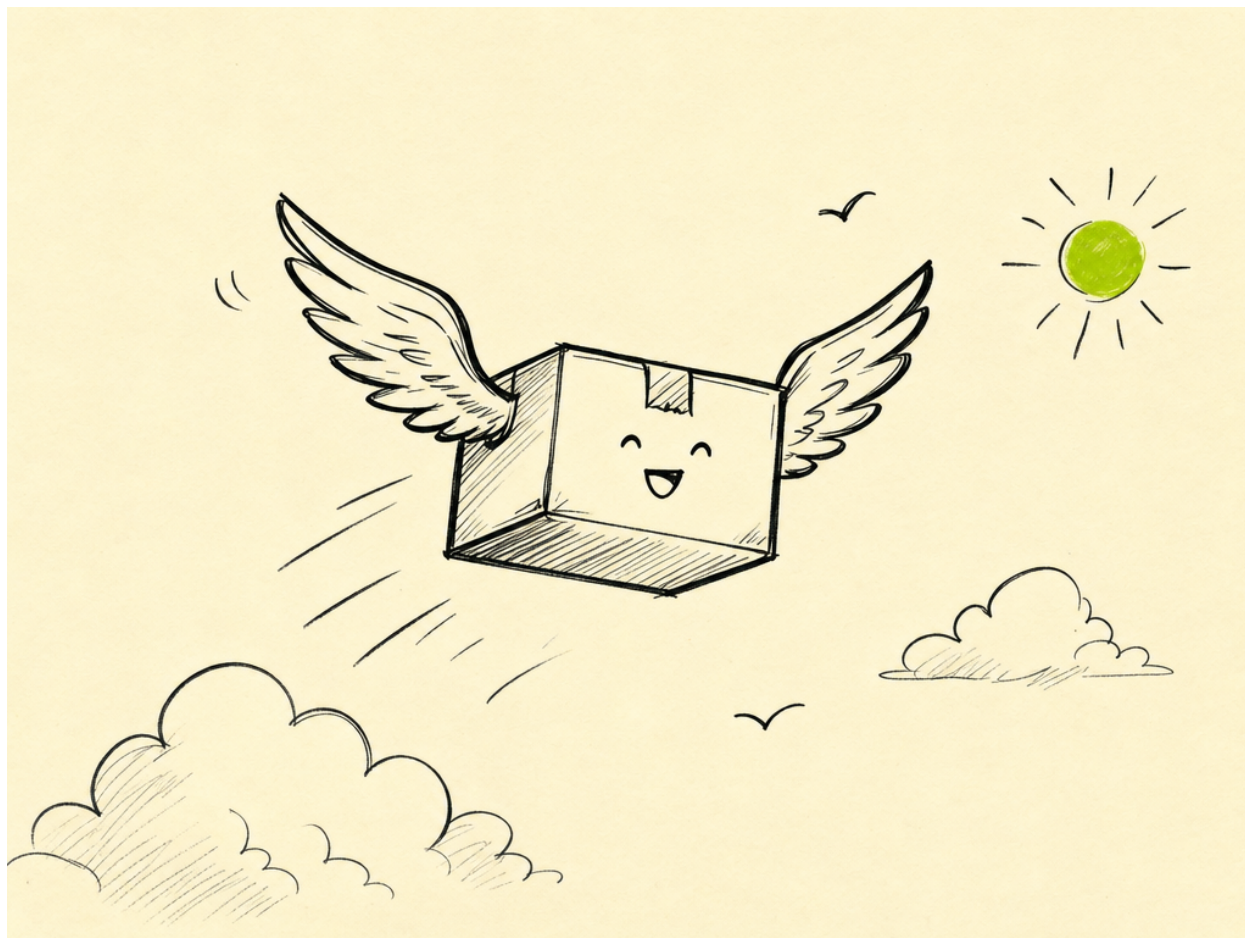
- Guardrails são o que permite à IA correr enquanto uma rede de segurança segura as quedas — eles te aceleram, não te atrasam.
- Testes, verificação de tipos e linting capturam erros de graça; conecte-os ao CI para que as verificações rodem automaticamente.
- Limite as mudanças da IA a um escopo estreito e leia todo diff com espírito crítico antes de mesclar.
- Uma verificação vermelha é informação — faça a triagem (bug real / teste desatualizado / verificação instável) antes de deixar a IA “consertar”.
- Nunca lance uma mudança que você não leu.

Coloque em prática

Em um projeto seu, configure uma verificação automatizada que você ainda não tem — um único teste, um linter, ou uma etapa de verificação de tipos — e adicione-a a um workflow de CI para que rode a cada push. Depois, quebre algo pequeno de propósito e confirme que a verificação fica vermelha antes de corrigir.

```
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.
```

Colocando em Produção



Você tem construído. O app funciona na sua máquina, a demonstração impressionou um amigo, e a vontade de colocá-lo no mundo é real. Este é o momento em que muitos criadores solo travam — não porque o código está quebrado, mas porque “colocar no ar” parece uma disciplina diferente, com suas próprias artes obscuras: DNS (Domain Name System, sistema de nomes de domínio, o catálogo de endereços da internet que transforma um nome comoseudominio.com no número onde um servidor mora), servidores, segredos, webhooks de pagamento. A boa notícia: esse é exatamente o tipo de trabalho mecânico e bem documentado no qual a IA é melhor. Seu trabalho é dirigir. O trabalho dela é digitar os comandos.

Mas “dirigir” é a palavra que carrega o peso aqui. O tema deste livro inteiro é *entenda o que você entrega*. Você não precisa memorizar as flags (opções de linha de comando), mas precisa saber para que serve cada etapa — do que um webhook te protege, por que um segredo nunca pode ir parar no seu repositório, o que um limite de gasto está contendo. Quando você entende a forma da coisa, consegue perceber quando a resposta confiante da IA está sutilmente errada, e consegue se recuperar quando o primeiro deploy explode. Os criadores que se queimam no lançamento são quase sempre aqueles que colaram comandos que não conseguiam ler.

Este capítulo percorre toda a última milha, de um app funcionando localmente até uma URL (Uniform Resource Locator, o endereço da web que as pessoas digitam no navegador) real pela qual as pessoas podem te pagar.

Escolha um Host e Faça o Deploy

Para a maioria dos apps vibe-coded, você não precisa de um servidor para cuidar sozinho. Plataformas modernas pegam seu código e te dão uma URL ao vivo em minutos. As escolhas comuns:

- **Sites estáticos e frontends:** Cloudflare Pages, Vercel, Netlify, GitHub Pages.
- **Apps full-stack e APIs (Application Programming Interfaces, interfaces de programação de aplicações, as portas que um programa usa para conversar com outro):** Cloudflare Workers, Vercel, Render, Fly.io, Railway.
- **Backends mais pesados:** uma plataforma de containers ou um VPS pequeno (Virtual Private Server, servidor privado virtual, seu próprio pedaço alugado de um computador sempre ligado em um data center).

A jogada certa é contar para a IA o que você construiu e deixar que ela recomende. Tente: *“Este é um app Next.js com banco de dados Postgres. Quero o host confiável mais barato com um plano gratuito generoso. Me dê os passos de deploy e um arquivo de configuração.”* Seja específico sobre as coisas que realmente limitam a escolha: seu framework, se você tem um banco de dados, se precisa de jobs em segundo plano ou upload de arquivos, e seu orçamento. Uma página estática de marketing e um app full-stack com fila de processamento têm lares muito diferentes.

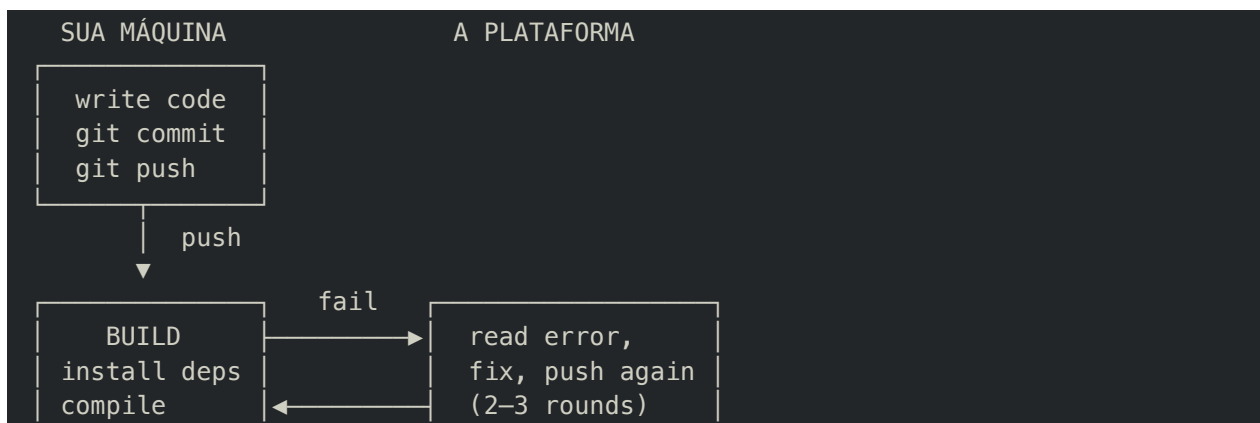
Depois de escolher um, o deploy geralmente é alguns comandos:

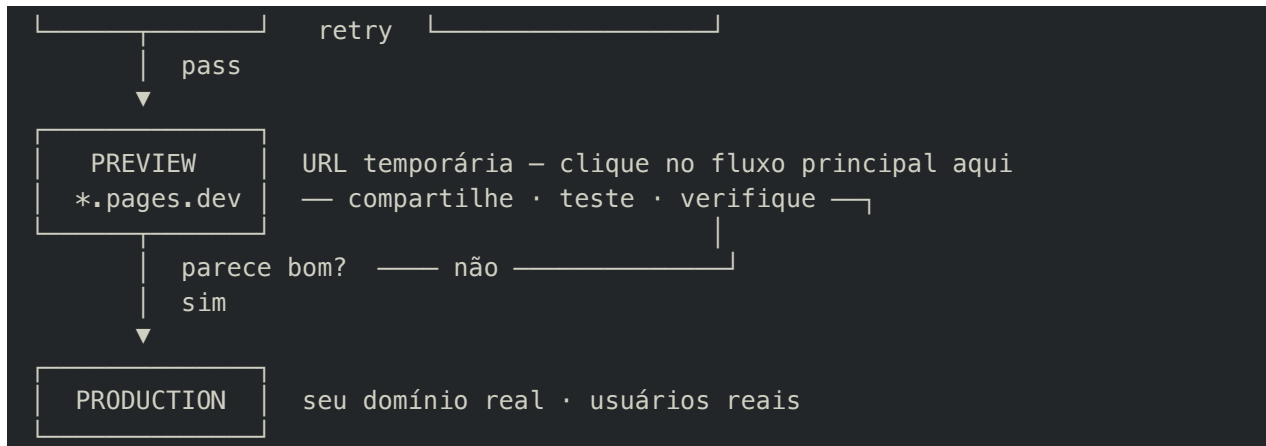
```
# Exemplo: Cloudflare Workers via Wrangler
npm install -g wrangler
wrangler login
wrangler deploy

# Exemplo: Vercel
npm install -g vercel
vercel --prod
```

Deixe a IA gerar a configuração da plataforma (wrangler.toml, vercel.json, etc.) e explicar qualquer erro que o primeiro deploy jogar na tela. Primeiros deploys quase sempre lançam algum erro — um comando de build faltando, um diretório de saída errado, uma versão do Node incompatível, uma variável de ambiente (uma configuração nomeada entregue ao seu app de fora do código, como uma chave debaixo do tapete que o app sabe onde procurar) que o build precisa mas não encontra. Nada disso significa que você fez algo idiota; significa que a plataforma precisa que você diga mais uma coisa. Cole o erro de volta, na íntegra, e peça a correção — e pergunte *por que* aconteceu, para que o próximo te surpreenda menos. Duas ou três rodadas e você está no ar em uma URL temporária. Abra essa URL e clique pelo fluxo principal antes de comemorar — “o deploy teve sucesso” e “o app funciona” não são a mesma afirmação.

Todo deploy percorre o mesmo pipeline. A etapa de preview é sua rede de segurança: ela dá a cada mudança sua própria URL para você clicar antes que ela possa sequer tocar em usuários reais.





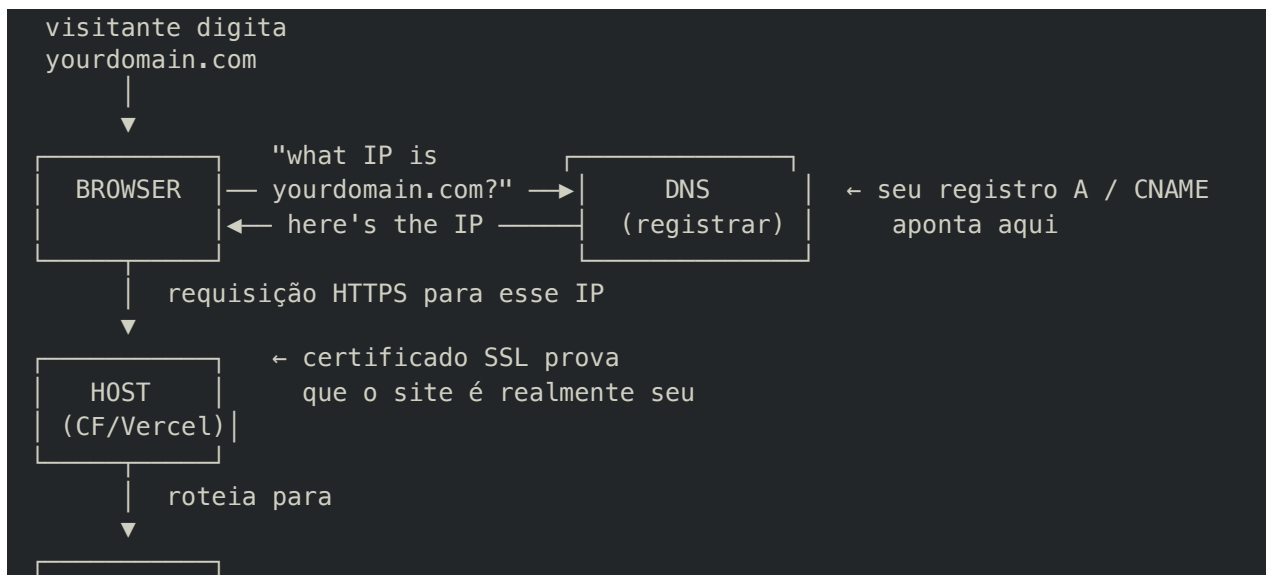
Domínios Personalizados e DNS

Uma URL `*.workers.dev` ou `*.vercel.app` serve para testar, mas você quer o seu próprio nome. O fluxo:

1. Compre o domínio (Namecheap, Cloudflare Registrar, Porkbun — escolha um).
2. Aponte o DNS dele para o seu host. Isso geralmente significa adicionar um registro CNAME ou A, ou trocar os nameservers (servidores de nomes) do domínio inteiramente.
3. Adicione o domínio no painel do seu host e deixe que ele emita um certificado SSL (a carteira de identidade digital que prova que o site é realmente seu e ativa o cadeado e o HTTPS — a versão segura e criptografada do protocolo de transferência da web) automaticamente.

Um modelo mental rápido para os registros pararem de parecer mágica: um **registro A** aponta um nome direto para um endereço IP; um **CNAME** aponta um nome para *outro nome* (útil quando o IP do host pode mudar sem você perceber); e os **nameservers** entregam o domínio inteiro a um provedor, para que ele gerencie cada registro para você. A maioria dos hosts modernos prefere um CNAME ou uma troca completa de nameservers — eles querem gerenciar o certificado SSL e o roteamento de borda por conta própria.

Aqui está toda a cadeia que a requisição de um visitante percorre, e onde cada peça que você configurou vive nela:



YOUR APP | serve a página

DNS é a parte mais confusa para iniciantes, principalmente por causa do jargão e dos atrasos de propagação. Deixe a IA traduzir. Pergunte: “*Meu domínio está registrado no Namecheap e estou hospedando no Cloudflare Pages. Me guie exatamente por quais registros adicionar, quais valores, e como verificar que funcionou.*” Depois verifique o resultado você mesmo com uma checagem rápida, em vez de apenas atualizar o navegador:

```
# Verifique os registros que o mundo realmente vê
dig yourdomain.com
nslookup yourdomain.com

# Confirme que o certificado está ativo e não expirado
curl -I https://yourdomain.com
```

Mudanças de DNS podem levar de minutos a um dia para propagar, e o cache do seu próprio navegador é o pior mentiroso de todos — ele vai te mostrar alegremente uma resposta desatualizada muito depois do resto da internet já ter atualizado. Se não estiver ativo imediatamente, isso é normal: verifique pelo seu celular usando dados móveis, ou use um verificador online de “propagação de DNS”, antes de supor que algo está quebrado.

Variáveis de Ambiente e Segredos

Seu código precisa de chaves de API, URLs de banco de dados e tokens. Isso **nunca** pode ser commitado no seu repositório. Uma vez que um segredo entra no histórico do git, ele é efetivamente público para sempre — apagar a linha em um commit posterior *não* o remove do histórico, e bots vasculham repositórios públicos novos em busca de chaves vazadas em minutos. Duas regras te levam longe:

- Mantenha os segredos em um arquivo `.env` local, e garanta que `.env` esteja no seu `.gitignore`.
- Configure os mesmos valores como variáveis de ambiente no painel do seu host (ou via CLI — Command-Line Interface, interface de linha de comando, a janela de prompt de texto onde você digita comandos em vez de clicar) para produção.

```
# .env (apenas local — nunca faça commit disso)
DATABASE_URL=postgres://localhost:5432/app
STRIPE_SECRET_KEY=sk_test_xxx
SESSION_SECRET=change-me

# Configure os equivalentes de produção no host
wrangler secret put STRIPE_SECRET_KEY
vercel env add STRIPE_SECRET_KEY production
```

Mantenha um `.env.example` commitado listando os *nomes* (não os valores) de cada variável, para que você-do-futuro e a IA saibam o que o app espera. Note o prefixo `sk_test_` acima — mantenha suas chaves de teste e de produção claramente distinguíveis, e nunca deixe uma chave de produção parada em um `.env` local que você poderia colar em um chat sem querer. Se você alguma vez colar um segredo em um chat ou commitar um por acidente, trate-o como comprometido e rotacione-o imediatamente: gere uma nova chave no painel do provedor, atualize-a em todos os lugares, e revogue a antiga. Rotacionar custa cinco minutos; uma URL de banco de dados vazada pode custar tudo o que está no banco.

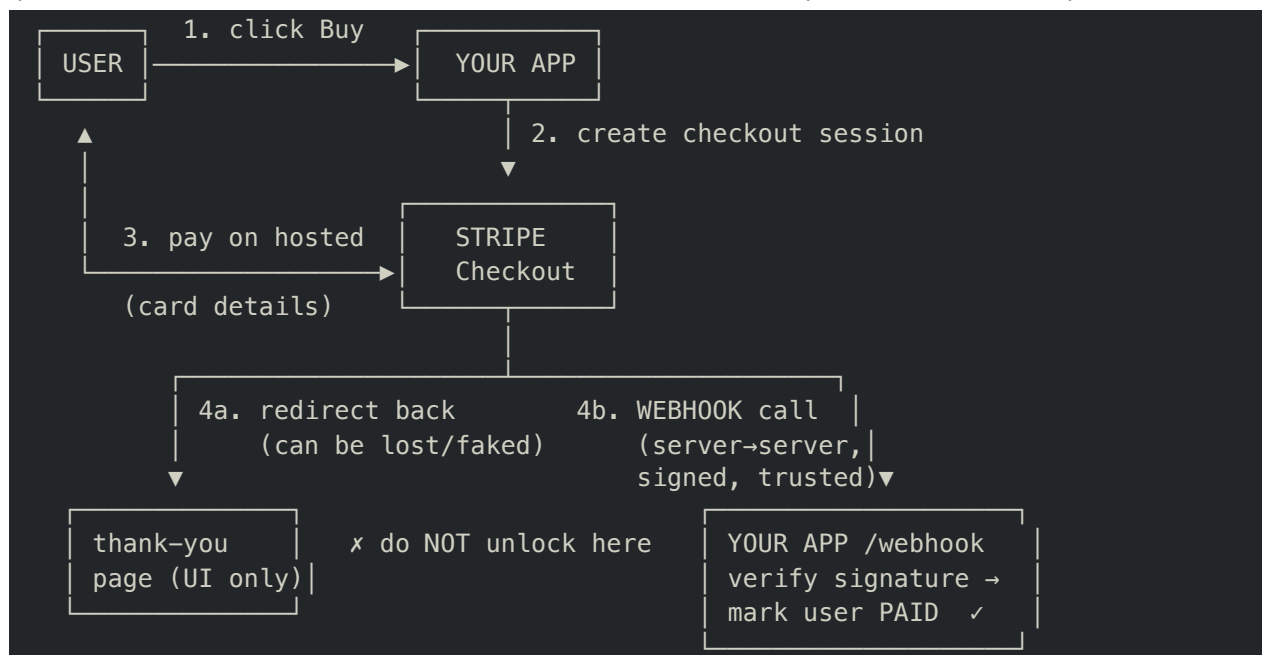
Configurando Pagamentos

Se você quer receber pagamentos, o caminho padrão para um criador solo é o Stripe (ou Lemon Squeezy / Paddle, se você quiser que eles cuidem do imposto sobre vendas para você — como merchant of record,

comerciante de registro, eles ficam com uma fatia maior, mas fazem a dor de cabeça fiscal desaparecer). A mecânica é repetitiva e bem documentada, o que os torna ideais para delegar. As peças de que você precisa:

- Um fluxo de **checkout** — o Stripe Checkout é o mais rápido; ele te entrega uma página de pagamento hospedada, então você nunca toca em dados de cartão crus.
- Pense em um webhook como a empresa de pagamento ligando para o seu app no instante em que uma venda é confirmada, em vez do seu app ficar perguntando “já foi processado?”. Um **webhook** é exatamente isso: um endpoint (uma URL no seu servidor) que o provedor de pagamento chama para que seu app saiba quando um pagamento realmente teve sucesso. Essa é a parte que iniciantes pulam e depois ficam se perguntando por que os pedidos nunca são entregues. O redirecionamento de volta para o seu site pode ser fechado, bloqueado ou falsificado; o webhook é a fonte da verdade servidor-a-servidor. Nunca confie apenas no redirecionamento; confie no webhook.
- Uma forma de **armazenar direitos de acesso** — marcar o usuário como pago no seu banco de dados, para que uma atualização de página ou um novo dispositivo ainda mostrem o que ele comprou.

O insight chave é *em qual* seta você confia. O redirecionamento do usuário de volta ao seu site pode se perder ou ser falsificado; o webhook é a linha servidor-a-servidor que realmente libera o produto:



Peça para a IA montar o esqueleto dos três e explicar a verificação da assinatura do webhook, já que essa é a única etapa sensível em termos de segurança — sem ela, qualquer um que encontrar a URL do seu webhook poderia falsificar um evento de “pagamento bem-sucedido” e liberar seu produto de graça. Teste tudo no modo de teste do Stripe com os números de cartão de teste deles (4242 4242 4242 4242 é o famoso) antes de trocar para as chaves de produção, e use a Stripe CLI para encaminhar eventos de webhook para sua máquina local, para que você possa observá-los disparando. Não pule o ensaio no modo de teste. O primeiro dinheiro real que passar pelo seu app deve ser uma transação que você já viu funcionar uma dúzia de vezes.

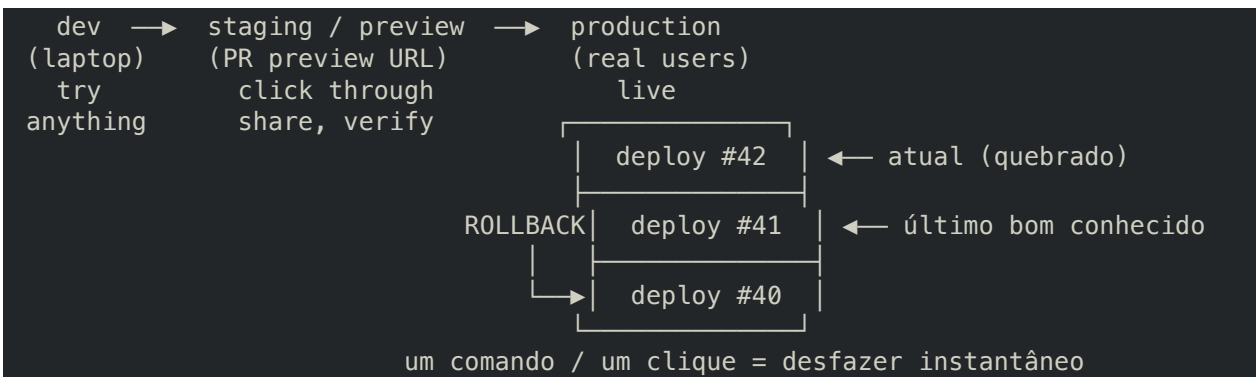
Staging e Rollbacks

Dois hábitos separam um lançamento que se recupera com graça de um que vira um pânico de meia-noite: testar mudanças em algum lugar seguro antes de elas chegarem aos usuários reais, e conseguir desfazer um deploy ruim instantaneamente.

Um **ambiente de staging** é apenas uma segunda cópia do seu app — mesmo código, URL separada,

banco de dados separado (ou descartável) — onde você testa coisas antes da produção. A maioria dos hosts torna isso quase gratuito: todo pull request ganha automaticamente sua própria **URL de preview**, para que você possa clicar por uma mudança e compartilhá-la com alguém antes de fazer o merge. Use isso especialmente para qualquer coisa que você não consiga reverter facilmente, como uma migração de banco de dados.

Imagine os três ambientes como uma escada que uma mudança sobe, com o rollback como a corda que puxa a produção de volta direto para o último degrau bom:



Um **rollback** é o seu botão de desfazer para o app inteiro. Como essas plataformas mantêm todo deploy anterior, voltar para a última versão boa geralmente é um comando ou um clique no painel:

```
# Cloudflare Pages: refazer o deploy de um build conhecido como bom
wrangler pages deployment list
wrangler pages deployment rollback <DEPLOYMENT_ID>

# Vercel: promover um deployment anterior de volta à produção
vercel rollback <DEPLOYMENT_URL>
```

Conheça esse comando *antes* de precisar dele, não durante a interrupção. Todo o sentido da velocidade do vibe coding é que entregar é barato — mas entregar barato só parece seguro quando desentregar é igualmente barato.

Monitoramento e Rastreamento de Erros

Assim que as pessoas usarem seu app, você precisa saber quando ele quebra — idealmente antes que elas te avisem. O kit inicial leve:

- **Rastreamento de erros:** o Sentry captura e agrupa exceções com stack traces, e te diz qual usuário e qual linha disparou cada uma. Poucas linhas para instalar.
- **Monitoramento de uptime:** um pinger gratuito (UptimeRobot, Better Stack) bate no seu site a cada poucos minutos e te manda um e-mail quando ele cai.
- **Logs:** saiba onde seu host os guarda (wrangler tail, o painel da Vercel, etc.) para que você possa investigar quando algo parecer errado.

Peça para a IA adicionar o Sentry e montar um endpoint básico de health-check (verificação de saúde) — uma rotinha pequena como /health que retorna 200 OK e confirma que o banco de dados está acessível, algo que seu monitor de uptime pode verificar. O objetivo não é observabilidade de nível empresarial — é simplesmente *não descobrir sobre uma interrupção por um tweet furioso*.

Performance e Cache

Apps lentos perdem usuários e ficam mal ranqueados em buscas. Você não precisa micro-otimizar no primeiro dia, mas pegue as vitórias baratas:

- Imagine manter cópias dos seus arquivos em armazéns espalhados pelo mundo todo, para que cada visitante seja atendido pelo mais próximo dele. Isso é uma **CDN** (Content Delivery Network, rede de entrega de conteúdo) — sirva recursos estáticos através de uma (a maioria dos hosts faz isso automaticamente).
- Faça **cache** de respostas caras e consultas de banco de dados que não mudam com frequência.
- Comprima e carregue imagens de forma preguiçosa (lazy-load); envie menos JavaScript.
- Rode o **Lighthouse** no Chrome DevTools e alimente o relatório para a IA priorizar as correções.

Pergunte: *“Aqui está meu relatório do Lighthouse. Me dê as três mudanças com o maior impacto pelo menor esforço.”* Deixe que a medição, não o achismo, guie o que você otimiza — é genuinamente comum passar um dia inteiro otimizando a coisa errada porque ela *parecia* lenta, quando o custo real era uma única consulta sem índice ou uma imagem de destaque de 4 MB que você esqueceu de comprimir.

Checklist Pré-Lançamento

Antes de anunciar qualquer coisa, percorra esta lista. Peça para a IA te ajudar a verificar cada item em vez de confiar na sua memória.

- ☐ O app faz deploy limpo e a URL de produção carrega
- ☐ O domínio personalizado resolve com HTTPS válido
- ☐ Todos os segredos configurados em produção; nenhum commitado no repositório
- ☐ `.env` está no gitignore e `.env.example` está atualizado
- ☐ Pagamentos testados de ponta a ponta (incluindo o webhook) em modo de teste
- ☐ Cadastro, login e o fluxo principal funcionam em produção
- ☐ Rastreamento de erros e monitoramento de uptime estão ativos
- ☐ Você sabe como fazer rollback, e já testou uma vez
- ☐ Layout mobile verificado em um celular real
- ☐ Uma política de privacidade básica e página de termos existem
- ☐ Backups do banco de dados estão ativados
- ☐ Uma forma clara dos usuários entrarem em contato com você (e-mail ou formulário)

Depois do Lançamento

Entregar é o começo, não o fim. A primeira versão é uma hipótese; usuários reais são o experimento. Observe seu painel de erros, leia cada mensagem, e entregue pequenas correções rápido. Resista ao impulso de reconstruir tudo no primeiro sinal de atrito — corrija as arestas afiadas, dobre a aposta no que as pessoas realmente usam, e deixe o resto esperar.

O hábito mais poderoso aqui é um loop apertado: perceba um problema ou uma ideia, descreva para a IA, entregue a mudança, observe o resultado. Como o trabalho mecânico é delegado, você pode iterar em horas em vez de semanas. Essa velocidade — não nenhuma funcionalidade isolada — é a vantagem real de construir por vibe. Você está no ar. Agora continue.

Recapitulação e Prática

Principais aprendizados

- Entregar é um fluxo de trabalho, não um momento: faça deploy, observe, corrija, repita — e transforme o rollback em algo que você já testou uma vez.
- Rode a checklist pré-lançamento antes de anunciar qualquer coisa, e peça para a IA verificar cada item em vez de confiar na sua memória.
- Deixe a medição guiar a otimização — um relatório do Lighthouse vence um palpite sobre o que “parece lento” todas as vezes.
- Segredos vivem na configuração de produção, nunca no repositório; `.env` está no gitignore e `.env.example` permanece atualizado.

- A primeira versão é uma hipótese; usuários reais são o experimento, então itere em correções pequenas e rápidas.

Coloque em prática

Pegue um app que você já entregou (ou entregue um pequeno agora) e rode a checklist pré-lançamento completa contra ele, linha por linha. Para cada caixa que você não consegue marcar honestamente, cole o item que falhou para a IA e peça a menor mudança que fecha a lacuna. Termine clicando em um link profundo no site ao vivo — não apenas na página inicial — para confirmar que a produção se comporta como o dev.

Prompt do 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.
```

Estudos de Caso: Do Zero ao Lançamento



Teoria é útil, mas nada ensina a vibe coding como ver acontecer. Este capítulo percorre quatro projetos pequenos, mas reais, construídos e lançados do início ao fim dirigindo uma IA. Nenhum deles é um brinquedo. Cada um tem usuários, um domínio próprio, e dinheiro ou tempo em jogo.

Para cada um, você verá a ideia, a especificação que demos à IA, a stack (conjunto de tecnologias) que escolhemos, os prompts que realmente moveram o ponteiro, os obstáculos que quase nos derrubaram, e como ele foi ao ar. Leia isso como manuais de operação, não como um “melhores momentos”. O importante não é que esses projetos exatos importem — é que os *movimentos* se repetem. No quarto caso você já vai reconhecer o ritmo: uma especificação curta, uma stack gerenciada, depuração guiada por evidências, e um lançamento que testa contra a realidade.

Estudo de Caso 1: Uma Página de Vendas Multilíngue com Pagamentos

A ideia

Uma amiga vende um guia em PDF de US\$ 29 para estrangeiros se mudando para a Coreia. Ela queria uma única página que carregasse rápido no mundo todo, falasse inglês, coreano e chinês, e recebesse pagamentos com cartão sem que ela precisasse tocar em um servidor.

A especificação

Mantivemos a especificação em um único parágrafo antes de escrever qualquer código:

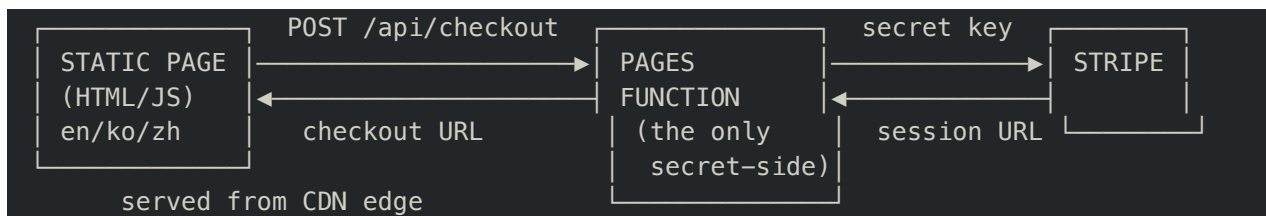
```
Build a single-page marketing site for a $29 digital product.
Three languages (en/ko/zh), auto-detected from the browser but
switchable with a toggle. One "Buy" button that opens Stripe
Checkout and redirects to a thank-you page on success. No backend
server I have to maintain. Must load fast from anywhere.
```

Note o que a especificação *não* diz: nenhum framework, nenhum design system, nenhum analytics, nenhuma captura de e-mail. Cada um desses seria um recurso razoável, e cada um teria atrasado o primeiro lançamento. A disciplina de uma especificação de um parágrafo é, na maior parte, a disciplina de deixar coisas de fora.

A stack

“Nenhum backend que eu mantenho” mais “rápido no mundo todo” apontava direto para hospedagem estática em uma CDN de borda (Content Delivery Network — cópias dos seus arquivos guardadas em data centers pelo mundo todo, para que cada visitante carregue a partir de um próximo) com uma pequena função serverless para a única coisa que *precisa* de um segredo: criar a sessão do Stripe. Escolhemos um site estático no Cloudflare Pages com uma única Pages Function. Toda a arquitetura cabe em uma frase — uma página estática que chama uma função que conversa com o Stripe — e essa simplicidade foi o que tornou tudo depurável mais tarde.

O sistema inteiro é três caixas, e só a do meio toca em algum segredo:



Os prompts principais

Começamos amplo e deixamos a IA propor a estrutura:

```
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.
```

Depois o caminho de pagamento, onde mora o segredo:

```
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.
```

A frase “never hardcode it” (nunca deixe fixo no código) merece seu lugar. Se deixada por conta própria, uma IA vai facilmente colocar uma chave de exemplo direto no código para fazer o exemplo “funcionar”, e uma chave de exemplo tem o hábito de virar uma chave real que vai parar em produção. Dizer isso uma vez, em voz alta, no prompt, sai mais barato do que pegar isso na revisão.

O obstáculo

O primeiro teste ao vivo falhou: o botão de comprar não fazia nada, e o console do navegador mostrava um erro de CORS (Cross-Origin Resource Sharing — a regra do navegador sobre quais sites podem chamar

quais outros sites). Não ficamos adivinhando. Colamos o erro exato de volta:

```
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?
```

A IA identificou na hora: a função estava retornando a URL do Stripe como JSON (JavaScript Object Notation — um formato de texto simples e legível que programas usam para trocar dados estruturados), mas o fetch estava sendo feito antes de o preflight OPTIONS da função ser tratado, porque tínhamos implantado a página estática e a função em dois projetos diferentes do Pages por engano. O conserto de verdade era sobre a topologia de deploy, não sobre código. Movemos a função para o diretório /functions do mesmo projeto e o erro desapareceu.

Vale a pena parar nessa lição: o *sintoma* estava no código (um erro de CORS vindo de um fetch), mas a *causa* estava na infraestrutura (dois projetos em vez de um). Se tivéssemos deixado a IA “consertar o erro de CORS” sem colar o contexto de configuração, ela teria parafusado cabeçalhos Access-Control-Allow-Origin — um conserto que compila, disfarça o sintoma, e deixa o bug de verdade vivo. Cole o erro literal e o contexto ao redor, e não aceite a primeira explicação plausível se ela não bater com a sua configuração.

O lançamento

Adicionamos as chaves reais do Stripe como variáveis de ambiente criptografadas no painel do Pages (nunca no repositório), apontamos o DNS (Domain Name System, a agenda de endereços da internet que mapeia um nome para um servidor) do domínio dela para o Pages, e fizemos uma compra de teste real de US\$ 29 com cartão, e depois reembolsamos. Também clicamos de propósito no caminho de falha — um cartão de teste recusado — para confirmar que ela não foi cobrada e que apareceu uma mensagem sensata em vez de uma tela em branco. No ar em uma tarde. Ela fez sua primeira venda naquela mesma noite.

Estudo de Caso 2: Uma Pequena Ferramenta SaaS com Autenticação e Banco de Dados

A ideia

Uma designer freelancer queria um painel privado para registrar horas cobráveis por cliente e exportar um CSV (Comma-Separated Values — um arquivo de planilha em texto simples que qualquer programa de planilhas consegue abrir) mensal. Nada sofisticado, mas precisava de contas (para que os dados dela fossem só dela) e persistência.

A especificação

Autenticação e um banco de dados elevam a aposta, então a especificação ficou mais específica sobre *limites*:

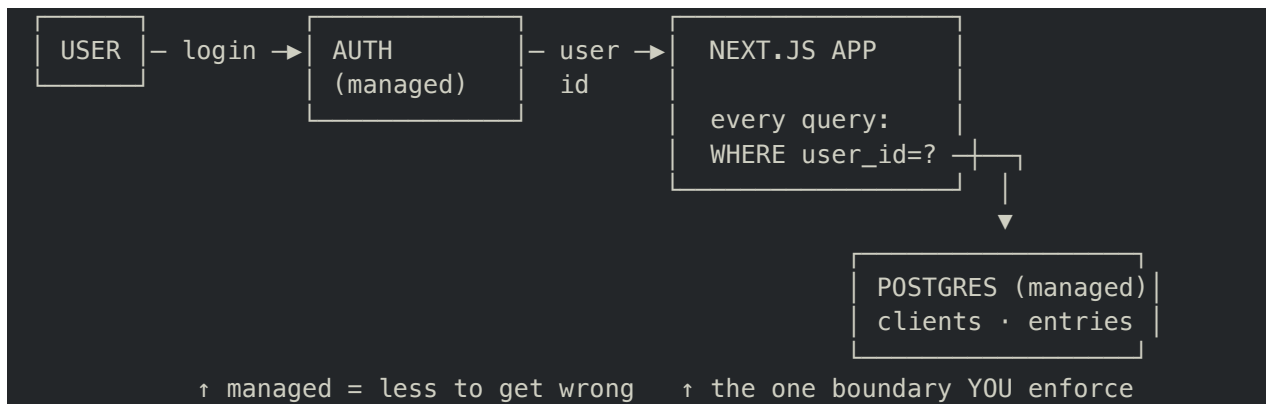
```
A logged-in web app where a user can: sign up / log in with email,
create clients, log time entries (date, client, hours, note),
see a table of entries filtered by month, and download that month
as CSV. Each user only ever sees their own data. Mobile-friendly.
```

“Each user only ever sees their own data” (cada usuário só vê seus próprios dados) parece uma linha de experiência do usuário. Na verdade é o modelo de segurança de todo o app, comprimido em poucas palavras. Nomear isso na especificação significou que podíamos apontar de volta para ela sempre que a IA se desviasse.

A stack

Para quem constrói sozinho, a jogada vencedora é uma stack em que autenticação e banco de dados são serviços gerenciados, não código que você escreve. Escolhemos um app Next.js implantado em uma hospedagem serverless, com um banco de dados Postgres hospedado e um provedor de autenticação plug-and-play que cuida de login por e-mail, sessões e redefinição de senha por nós. Menos código para dar errado é menos código que uma IA pode errar em nosso nome. Autenticação em particular é uma categoria que você quase nunca quer construir do zero com uma IA: os modos de falha são silenciosos, o raio de impacto é a conta de todo mundo, e um provedor gerenciado já teve seus casos extremos testados por milhões de logins.

As peças gerenciadas ficam por fora; a única regra que você é dono fica no meio — toda consulta filtra por `user_id`, para que um usuário nunca consiga ler as linhas de outro:



Os prompts principais

Deixamos que o próprio template do provedor de autenticação fizesse o trabalho pesado, e depois direcionamos a IA a colocar a lógica de domínio por cima:

```
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.
```

A linha “MUST filter by user_id” foi a frase mais importante de todo o projeto. Repetimos essa restrição em quase todo prompt que tocava em dados, porque o bug mais assustador em um app multiusuário é um usuário ver as linhas de outro. A repetição parece redundante enquanto você a digita; é exatamente essa redundância que salva você, porque o modelo não tem memória de o quanto essa restrição importa entre prompts separados.

Para a exportação:

```
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.
```

O obstáculo

Durante os testes, criamos duas contas e descobrimos que a conta B conseguia ver os clientes da conta A em um menu suspenso. Esse é exatamente o bug que tínhamos. Em vez de pedir para a IA “consertar”, fizemos com que ela primeiro comprovasse o 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.
```

A auditoria trouxe à tona uma consulta — o carregador do menu suspenso — que tinha sido escrita antes de adicionarmos a restrição e escapou pelas frestas. Pedimos que ela adicionasse o filtro que faltava, e depois pedimos uma proteção:

```
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.
```

Isso transformou um conserto pontual em uma garantia estrutural. Depois tornamos a garantia testável, porque uma proteção que você não consegue verificar é só uma esperança:

```
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.
```

A lição: quando uma IA introduz um bug de segurança, não fique apenas consertando a instância — direcione-a a remover a *categoria* do erro, depois tranque a categoria com um teste que falha ruidosamente se alguém reabrir a brecha.

O lançamento

Semeamos um mês de teste com dados, exportamos o CSV, abrimos em uma planilha para confirmar que os números e a codificação estavam certos, depois definimos uma senha forte para o banco de dados e rotacionamos as credenciais para fora de qualquer arquivo local. Implantamos na hospedagem serverless, adicionamos as variáveis de ambiente de produção no painel dela, e demos a URL para ela. Ela mesma se cadastrou com um cadastro real. A construção inteira levou um fim de semana.

Estudo de Caso 3: Um Utilitário Pessoal de Automação

A ideia

Um puramente egoísta: um script que toda manhã pega os eventos da agenda do dia e o clima, monta um resumo de uma linha, e manda para um chat pessoal no Telegram. O objetivo era parar de abrir três apps antes do café.

A especificação

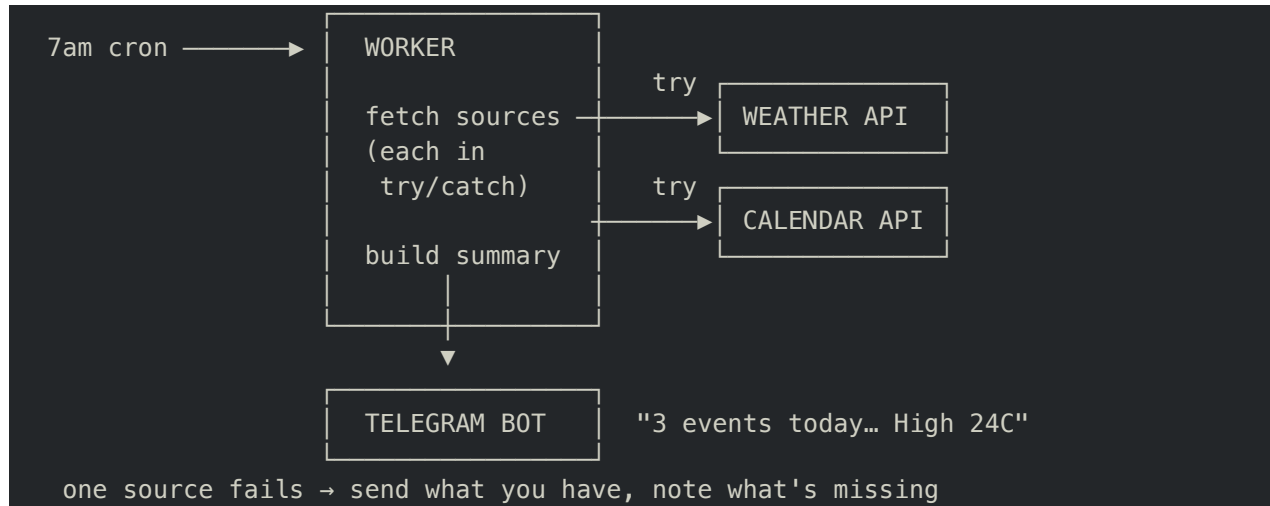
```
A scheduled job that runs at 7am my timezone. It reads today's events from my calendar, fetches the forecast for my city, and sends one Telegram message like: "3 events today, first at 9:30. High 24C, rain after 4pm." If any source fails, still send what you have and note what's missing.
```

Aquela última frase — degradar graciosamente em vez de quebrar — é o tipo de comportamento que você tem que pedir explicitamente. A IA não vai assumir isso sozinha. Se não for dito, o padrão que um modelo escolhe é “lançar erro”, o que para um job de cron às 7h significa uma manhã silenciosa e nenhuma pista do motivo.

A stack

Uma única função serverless agendada (um Worker acionado por Cron) para que não haja máquina nenhuma para manter rodando. A coisa inteira é um arquivo mais um agendamento. Para um utilitário pessoal isso importa mais do que parece: qualquer coisa com um servidor para cuidar é algo que você eventualmente vai esquecer de pagar, e o projeto morre por negligência em vez de por falha.

Um gatilho se ramifica em duas fontes e converge de volta em uma mensagem. O try/catch ao redor de cada fonte é o que permite que uma única falha degrade graciosamente em vez de matar toda a execução:



Os prompts principais

```
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.
```

Quando quisemos iterar no texto sem esperar até as 7h:

```
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.
```

Esse gatilho de teste nos salvou de um ciclo de feedback brutal — podíamos ver o resultado sob demanda em vez de uma vez por dia. É um movimento genérico que vale a pena roubar: sempre que seu código roda em um agendamento, construa para si mesmo um jeito de rodá-lo *agora*, sob demanda, antes de se comprometer a esperar pelo relógio.

O obstáculo

A mensagem do Telegram chegava, mas os horários estavam errados: os eventos apareciam em UTC, não no horário da Coreia. O clima estava certo. Isolamos o problema:

```
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.
```

A IA explicou que a API do clima (Application Programming Interface — a porta que um serviço expõe para que outro programa busque seus dados) já retornava strings localizadas, enquanto a API do calendário retornava carimbos de data/hora em UTC que estávamos imprimindo crus. Ela adicionou uma única conversão de fuso horário na etapa de formatação do calendário. Restringir o conserto a “apenas o caminho do calendário” impediu que ela reescrevesse o código do clima que já funcionava — um jeito comum de conserto de IA quebrar coisas que estavam boas. A segunda metade daquele prompt — “tell me why the weather path was already right” — fez dupla função: nos deu a causa raiz real em vez de um chute, e forçou a IA a articular a diferença para que seu conserto não “harmonizasse” acidentalmente os dois caminhos em um só, quebrado.

O lançamento

“Lançamento” aqui só significou implantar o Worker, definir o agendamento de cron, e adicionar os tokens de API como segredos criptografados. Batemos no endpoint `?test=1` algumas vezes para confirmar que a mensagem lia bem, e então deixamos correr. Está rodando toda manhã desde então — e a linha de degradação graciosa provou seu valor um mês depois, quando a API do clima teve uma queda e a mensagem ainda assim chegou, só sem a previsão.

Estudo de Caso 4: Um Site de Conteúdo que Você Não Precisa Cuidar

A ideia

Um hobbista queria transformar uma pasta de notas em Markdown — resenhas de restaurantes de anos de viagens — em um site público pesquisável, sem CMS para fazer login e sem mensalidade. Adicionar uma nota, dar push, pronto.

A especificação

```
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.
```

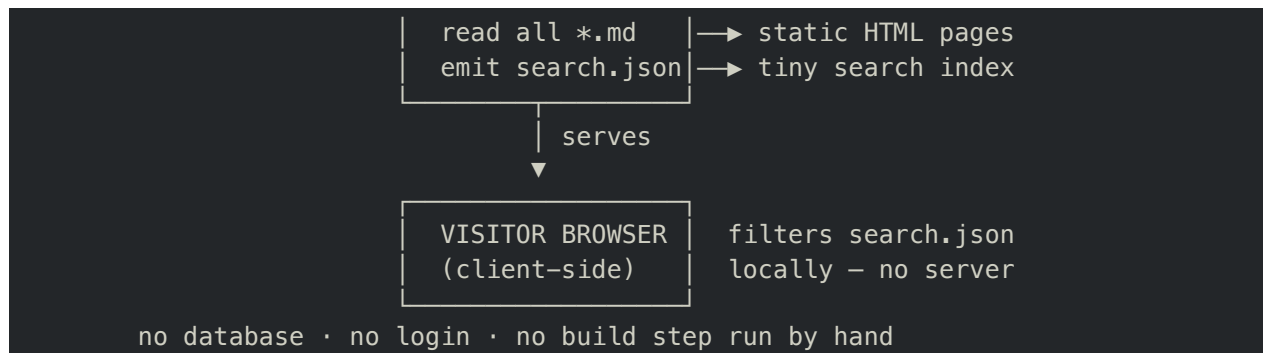
O requisito escondido está na última frase. “No build step I have to run by hand” (nenhuma etapa de build que eu tenha que rodar na mão) significa que a especificação é na verdade sobre o *fluxo de trabalho*, não só sobre o resultado — e requisitos de fluxo de trabalho são os que as IAs pulam a menos que você os nomeie, porque eles não aparecem no app em funcionamento.

A stack

Um gerador de site estático que lê Markdown no momento da build, implantado em uma hospedagem CDN com builds acionadas por Git. Push para o repositório, a hospedagem reconstrói, a CDN serve. A busca é do lado do cliente, sobre um pequeno índice JSON gerado na build, então não há servidor nem custo de consulta — uma boa troca com algumas centenas de resenhas, e uma troca deliberada que nomeamos em vez de descobrir.

Todo o fluxo de trabalho é uma única flecha: um push aciona uma build que publica. Nada roda no momento da requisição, exceto o próprio navegador do visitante filtrando um índice:

```
add review.md
git push → [ CDN HOST
              build step: ]
```



Os prompts principais

Ancoramos a IA na forma dos dados primeiro, antes de qualquer interface:

```
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 qualquer estilo) é uma alavanca deliberada. Peça a coisa toda de uma vez e você recebe uma página bonita conectada a dados nos quais você não pode confiar; peça a camada de dados primeiro e você pode verificar a base antes de um único pixel te distrair. Depois a busca, mantida propositalmente 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.
```

O obstáculo

A build passava localmente, mas o site implantado dava 404 em cada página de resenha individual. A página inicial funcionava; as páginas de resenha, não. Colamos o sintoma e a configuração em vez de ficar teorizando:

```
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.
```

Apontamos o diretório de saída para o que a hospedagem realmente implantava, as páginas apareceram, e adicionamos uma linha à checklist de lançamento: *clique em um link profundo no site ao vivo, não só na página inicial*. A lição mais ampla é que “funciona em dev” e “funciona implantado” são afirmações

diferentes, e a exportação estática é exatamente onde elas divergem — servidores de desenvolvimento são tolerantes com rotas de um jeito que hospedagem baseada em arquivos reais nunca é.

O lançamento

Fizemos push de uma resenha real, vimos a hospedagem reconstruir, e carregamos um link profundo para aquela resenha específica no domínio ao vivo — exatamente o caso sobre o qual o dev tinha mentido. Depois pesquisamos pela cidade dela para confirmar que o índice a tinha captado. Adicionar a *próxima* resenha foi um commit de uma linha, que era todo o objetivo: o lançamento não foi um momento, foi um fluxo de trabalho que continua funcionando sem nós.

O que os quatro têm em comum

- **A especificação veio primeiro, e era curta.** Um parágrafo, não um documento. Mas ela nomeou as restrições que importavam — idiomas, “só vê seus próprios dados”, “ainda assim envie o que você tem”, “fazer push de um arquivo deve publicá-lo” — incluindo as restrições de fluxo de trabalho que nunca aparecem no app em funcionamento.
- **Segredos nunca tocaram o repositório.** Todo projeto lia chaves de variáveis de ambiente ou segredos criptografados, definidos em um painel, não no código. Dissemos “nunca deixe fixo no código” em voz alta, porque o padrão é o oposto.
- **A stack foi escolhida para que houvesse menos coisa para dar errado.** Autenticação gerenciada, banco de dados gerenciado, hospedagem estática, funções serverless — cada escolha removeu uma categoria de código que a IA poderia quebrar em nosso nome, e uma categoria de coisa que teríamos que manter.
- **Obstáculos foram resolvidos alimentando a IA com evidências exatas** — o erro literal, o código real, a configuração de deploy, o sintoma verdadeiro — e restringindo os consertos de forma estreita para que o código que funcionava continuasse funcionando. Duas vezes o sintoma estava no código, mas a causa estava na infraestrutura.
- **Lançamento significou um teste real contra a realidade:** uma compra ao vivo (e uma recusada), uma segunda conta de usuário, um gatilho sob demanda, um link profundo no site implantado. Não “compilou”.

Você não precisa de uma grande ideia para começar. Você precisa de uma pequena, de uma especificação de um parágrafo, e da vontade de dirigir em vez de digitar — e da paciência de testar a coisa contra a realidade antes de chamá-la de pronta.

Recapitulação e Prática

Principais aprendizados

- A especificação veio primeiro e era curta — um parágrafo que nomeou as restrições que importavam, incluindo restrições de fluxo de trabalho que nunca aparecem no app em funcionamento.
- Segredos nunca tocaram o repositório; todo projeto lia chaves de variáveis de ambiente ou configuração criptografada definida em um painel.
- A stack foi escolhida para que houvesse menos coisa para dar errado — autenticação gerenciada, banco de dados gerenciado, hospedagem estática, funções serverless.
- Obstáculos foram resolvidos alimentando a IA com evidências exatas — o erro literal, o código real, a configuração de deploy — e restringindo os consertos de forma estreita.
- Lançamento significou um teste real contra a realidade, não “compilou”: uma compra ao vivo, uma segunda conta, um link profundo no site implantado.

Tente você mesmo

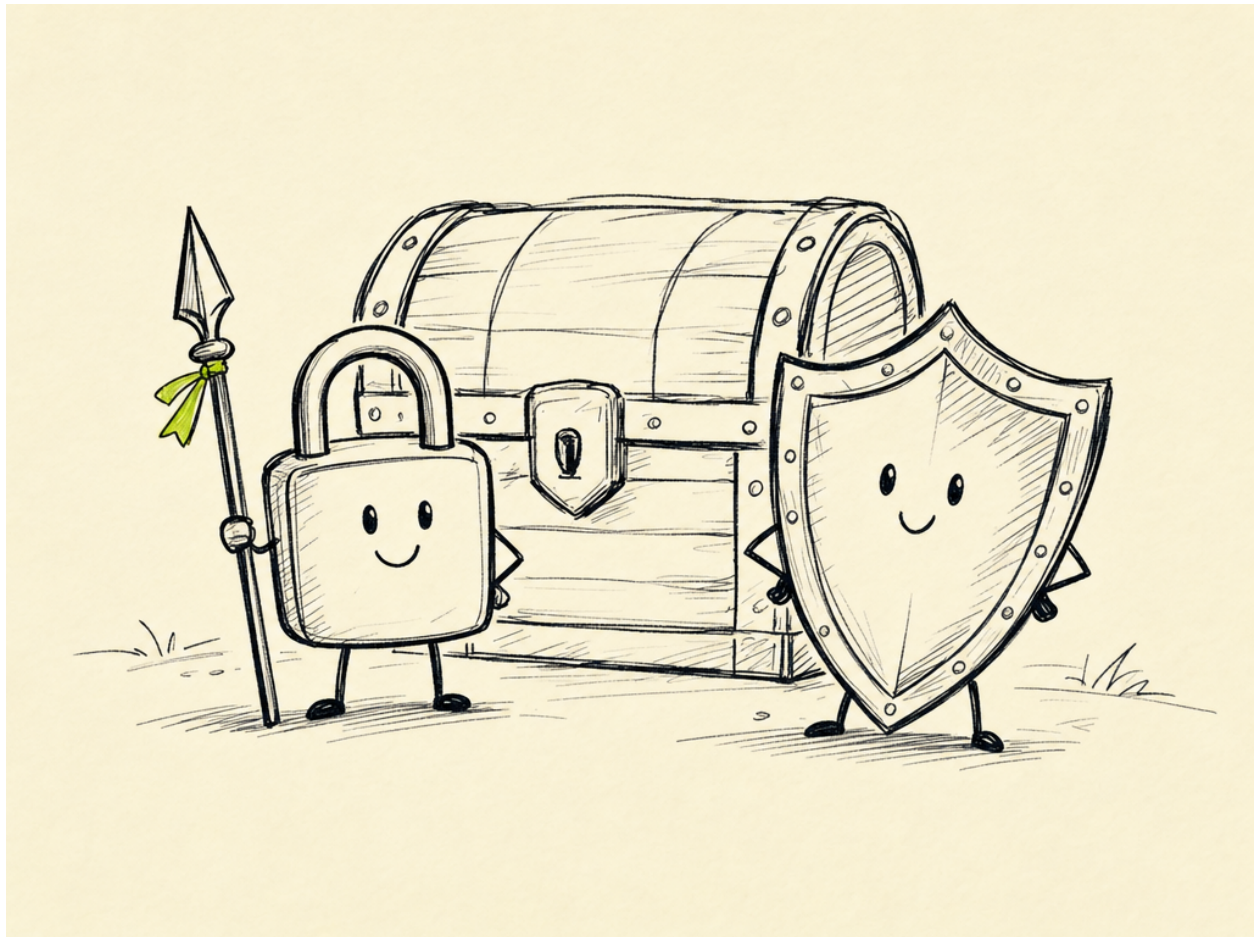
Escolha uma das suas próprias ideias e escreva a especificação de um parágrafo do jeito que esses quatro fizeram: nomeie a ação central, os dados que cada usuário pode ver, e pelo menos uma restrição de fluxo

de trabalho (como o conteúdo é publicado, quem pode fazer o quê). Depois liste os blocos de construção gerenciados nos quais você se apoiaria, para que houvesse menos código personalizado para quebrar. Mantenha isso em um único parágrafo mais uma lista curta de itens — se crescer além disso, você está especificando demais.

Prompt do 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.
```

Segurança para vibe coders



Segurança é a parte do vibe coding que morde em silêncio. Uma funcionalidade quebrada aparece na

hora — o botão não funciona, você conserta. Uma falha de segurança nunca aparece, até o dia em que alguém a encontra e despeja os dados dos seus usuários em um fórum. Nessa hora, a IA que escreveu o bug já se foi há muito tempo, e é você quem vai explicar para mil pessoas por que a senha delas vazou.

Este capítulo não é sobre medo. Você não precisa virar um pentester (profissional que testa sistemas simulando ataques) nem decorar a OWASP Top 10 (lista das dez falhas de segurança mais críticas em aplicações web) para lançar um app pequeno com segurança razoável. Você precisa entender um punhado de formas comuns pelas quais softwares são invadidos, reconhecer os padrões que a IA insiste em produzir, e adicionar um portão de revisão antes de lançar. A filosofia é a mesma do resto do livro: **nunca lance algo que você não entende** — e segurança é onde não entender custa mais caro.

Por que código gerado por IA é inseguro por padrão

A IA otimiza para o prompt que você deu a ela, e o seu prompt quase sempre diz “faça funcionar”, nunca “faça com segurança”. Então ela produz o caminho mais curto até uma funcionalidade que funciona. O caminho mais curto costuma ser o inseguro: consultas montadas por concatenação de strings, um endpoint (uma das portas endereçáveis do seu app — uma única URL onde o app responde a requisições) sem verificação de permissão, um segredo colado direto no código porque foi esse o exemplo que ela aprendeu.

Piora porque a IA aprendeu com a internet inteira — incluindo uma década de tutoriais e repositórios abandonados que já eram inseguros. Código inseguro é *mais* comum nos dados de treinamento dela do que código seguro, porque código seguro é mais difícil de escrever e mais raro de publicar. Quando você pede “um formulário de login”, estatisticamente você recebe o formulário de login médio da internet, e o formulário de login médio da internet tem problemas.

A IA também não tem noção de ameaça. Um humano construindo um upload de arquivo sente um lampejo de preocupação — *e se alguém enviar algo malicioso?* A IA não sente nada. Ela escreve o manipulador com a mesma confiança, esteja ele blindado ou escancarado. Não existe hesitação, nenhum “hmm, essa parte me deixa nervoso”. Você precisa fornecer o nervosismo sozinho.

A lição não é “código de IA é perigoso, não use”. É que **“funciona” e “é seguro” são duas perguntas diferentes**, e a IA só responde a primeira, a menos que você force ela a responder a segunda.

Injeção: quando a entrada vira código

Injeção é o bug sério mais antigo e ainda o mais comum. Acontece quando dados de um usuário são tratados como *instruções* em vez de *dados*. Duas variantes importam para a maioria dos apps.

Injeção de SQL — SQL (Structured Query Language, a linguagem que os apps usam para pedir dados a um banco de dados) — acontece quando a entrada do usuário se infiltra em uma consulta ao banco. O padrão vulnerável clássico:

```
// VULNERÁVEL: a entrada do usuário é colada direto na consulta
app.get("/user", (req, res) => {
  const name = req.query.name;
  db.query(`SELECT * FROM users WHERE name = '${name}'`);
});
// Se alguém passar name = '; DROP TABLE users; --
// sua consulta vira um comando para apagar a tabela.
```

A correção é uma **consulta parametrizada** (também chamada de prepared statement). Você envia a consulta e os dados separadamente, então o banco nunca confunde um com o outro:

```
// SEGURO: o valor é passado como parâmetro, nunca como código
app.get("/user", (req, res) => {
  const name = req.query.name;
```

```
db.query("SELECT * FROM users WHERE name = ?", [name]);
});
```

A regra: **nunca monte uma consulta por concatenação de strings**. Se você ver crases ou + colando SQL em torno de uma variável em um diff, pare e peça a versão parametrizada.

Os dois caminhos colocam a mesma entrada do usuário no banco de dados, mas só um impede que o banco confunda essa entrada com comandos:



XSS (cross-site scripting) é a mesma ideia no navegador. Se você pega a entrada do usuário e a coloca em uma página como HTML bruto (HyperText Markup Language, o código que diz ao navegador o que exibir), um atacante pode injetar uma tag `<script>` que roda no navegador dos seus outros usuários — roubando as sessões deles, por exemplo. A correção é o escaping (transformar caracteres especiais em texto seguro): renderizar o conteúdo do usuário como *texto*, não como HTML. Frameworks modernos como o React fazem escaping por padrão, o que é ótimo, até a IA recorrer a uma saída de emergência como `dangerouslySetInnerHTML` ou `innerHTML` para “fazer funcionar”. Isso contorna a proteção. Trate qualquer chamada desse tipo em um diff como algo a questionar.

O princípio unificador por trás de todo bug de injeção: **mantenha dados e código separados**. Sempre que a entrada do usuário cruza para dentro de uma consulta, um comando, um template ou HTML, algo precisa fazer escaping ou parametrizar.

Segredos e chaves de API: a armadilha da chave exposta

Um segredo é qualquer coisa que concede acesso: chaves de API (Application Programming Interface), senhas de banco de dados, tokens de provedores de pagamento, chaves de assinatura. O erro mais comum — e mais caro — do vibe coding é vaziar um deles.

Dois armadilhas pegam gente o tempo todo:

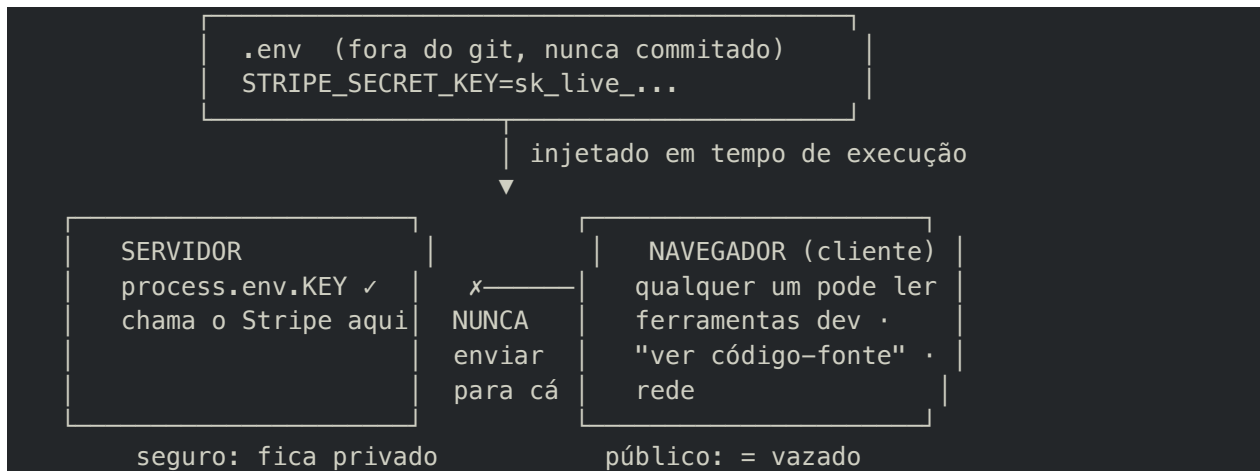
- **Segredos no código do cliente.** Qualquer coisa no seu frontend (o JavaScript que roda no navegador) é *pública*. Usuários podem abrir as ferramentas de desenvolvedor e ler isso. A IA, ao ser pedida para chamar uma API a partir de um componente React, vai colar sua chave secreta ali mesmo, feliz da vida — e agora qualquer pessoa que visite seu site pode copiá-la e inflar sua conta. Chaves secretas pertencem ao servidor, nunca ao código que vai para o navegador.
- **Segredos no repositório.** Uma chave fixada em um arquivo é enviada para o git. Mesmo que você a apague depois, ela vive para sempre no histórico do git, e bots vasculham repositórios públicos exatamente atrás disso minutos depois de um push.

O lar correto de um segredo é uma **variável de ambiente** — um valor fornecido ao app em tempo de execução, mantido totalmente fora do código:

```
// VULNERÁVEL: a chave está no código-fonte, será enviada ao git
const stripe = new Stripe("sk_live_51H8xQ2eZvK...");

// SEGURO: a chave é lida do ambiente, nunca escrita no código
const stripe = new Stripe(process.env.STRIPE_SECRET_KEY);
```

Um segredo tem exatamente um lar seguro — no servidor, lido do ambiente. O navegador é um lugar público: qualquer coisa enviada para lá pode ser lida por qualquer um que abra as ferramentas de desenvolvedor.



Adicione `.env` (e `.env.local`, etc.) ao seu `.gitignore` *antes* de escrever um único segredo. Se uma chave chegar a parar no seu código ou histórico, trate-a como queimada: gire-a (gere uma nova e revogue a antiga) — apagar a linha não basta, porque a chave antiga continua válida e continua por aí.

Autenticação vs autorização: o endpoint que qualquer um pode chamar

Essas duas palavras soam parecidas, e é na diferença entre elas que vivem as violações reais.

- **Autenticação** é *quem é você?* — fazer login, provar identidade.
- **Autorização** é *o que você tem permissão para fazer?* — se este usuário logado pode realizar esta ação sobre estes dados.

A IA é razoável em autenticação; bibliotecas cuidam da maior parte. Autorização é onde ela falha com frequência, porque autorização é específica das regras do seu app, e a IA não as conhece. O desastre clássico de livro-texto:

```
// VULNERÁVEL: verifica que você está logado, mas não QUEM você é
app.get("/admin/export-all-users", requireLogin, (req, res) => {
  res.json(db.getAllUsers()); // qualquer usuário logado pode acessar isso
});
```

Esse endpoint está “protegido” — você precisa estar logado. Mas *qualquer* usuário logado, incluindo um que se cadastrou trinta segundos atrás, pode chamá-lo e baixar os dados de todo mundo. Autenticação presente, autorização ausente. A correção é verificar a permissão sobre a própria ação:

```
// SEGURO: confirma que este usuário é de fato um administrador
app.get("/admin/export-all-users", requireLogin, (req, res) => {
  if (!req.user.isAdmin) return res.status(403).send("Forbidden");
```

```
res.json(db.getAllUsers());
});
```

Pense em dois portões que uma requisição precisa passar. Autenticação pergunta “você está logado?”; autorização pergunta “você tem permissão para fazer *isto*?” A violação acontece quando um app constrói o primeiro portão e esquece o segundo:



A mesma armadilha aparece em miniatura em todo lugar: um endpoint que devolve o pedido #1234 sem verificar se o pedido pertence ao usuário que fez a requisição. Qualquer um pode trocar o número na URL e ler o pedido de outra pessoa. A regra é sem graça e absoluta: **verifique autorização em todo endpoint que toca dados, e não confie em um ID vindo do cliente**. Não presuma que uma URL escondida é segura só porque está escondida — “ninguém sabe que isso existe” não é um controle de segurança.

Manipulação de arquivos e uploads

Permitir que usuários façam upload de arquivos é um campo minado silencioso, e a primeira versão da IA quase nunca fecha os buracos:

- **Valide o tipo e o tamanho.** Sem limites, alguém envia um arquivo de 10 GB ou um executável. Verifique o conteúdo real, não só a extensão do nome do arquivo, que qualquer um pode renomear.
- **Nunca confie no nome do arquivo.** Um nome como `../../../../etc/passwd` pode fazer um manipulador ingênuo escrever fora da pasta pretendida (isso se chama path traversal, ou travessia de diretório). Gere seus próprios nomes seguros em vez de usar o que foi fornecido.
- **Não sirva uploads a partir da mesma origem do seu app**, e não deixe arquivos enviados por usuários serem executados como código. Um arquivo fornecido pelo usuário que cai em uma pasta que seu servidor vai *executar* é um caminho direto para a invasão total.

Você não precisa construir tudo isso sozinho — um serviço de armazenamento cuida da maior parte — mas precisa *pedir*, porque a IA não vai oferecer isso por conta própria. “Adicione uma função de upload” te dá o caminho feliz. “Adicione uma função de upload que valide tipo e tamanho de arquivo, rejeite tudo que não seja imagem, e armazene os arquivos com nomes gerados” te dá algo mais próximo de seguro.

Risco de dependências: o pacote que a IA inventou

Apps modernos se apoiam em dezenas de pacotes de terceiros, e a IA os sugere livremente. Dois riscos andam junto.

Primeiro, **typosquatting (registro de nomes parecidos por engano de digitação) e pacotes alucinados**. Atacantes publicam pacotes maliciosos com nomes quase idênticos aos reais (requeusts em vez de requests), apostando num erro de digitação. A IA às vezes importa com confiança um pacote que não existe — e um atacante que percebe esse hábito pode registrar exatamente esse nome com malware dentro. Antes de instalar qualquer coisa que a IA sugerir, dê uma olhada: será que existe mesmo, tem números reais de download e um repositório, o nome está escrito do jeito que você esperaria?

Segundo, **dependências não verificadas em geral**. Todo pacote que você adiciona é código rodando com acesso total ao seu app. Mais dependências significa mais superfície tanto para bugs quanto para ataques de cadeia de suprimentos. Prefira poucas bibliotecas bem conhecidas em vez de uma longa cauda de pacotes obscuros, e pergunte “a gente realmente precisa de um pacote para isso, ou são dez linhas?” antes de adicionar um.

Rode a ferramenta de auditoria do seu ecossistema periodicamente (`npm audit`, `pip-audit`, e afins) — ela sinaliza vulnerabilidades conhecidas no que você já instalou, e a IA é boa em corrigir o que ela relata.

O portão de revisão de segurança

Aqui está o hábito único que transforma tudo isso, de uma preocupação, em um processo: antes de lançar, **faça a IA atacar o próprio código**. O modelo que escreveu a funcionalidade geralmente consegue encontrar os buracos nela — só que não vai fazer isso a menos que você peça. Vire a chave dele de construtor para adversário:

Você escreveu este endpoint. Agora aja como um atacante tentando quebrá-lo.

Liste todas as formas pelas quais um usuário malicioso poderia:

- ler ou modificar dados que não deveria (falhas de autorização)
- injetar código via entrada (injeção de SQL, XSS, injeção de comando)
- abusar de validação ausente ou limites de taxa

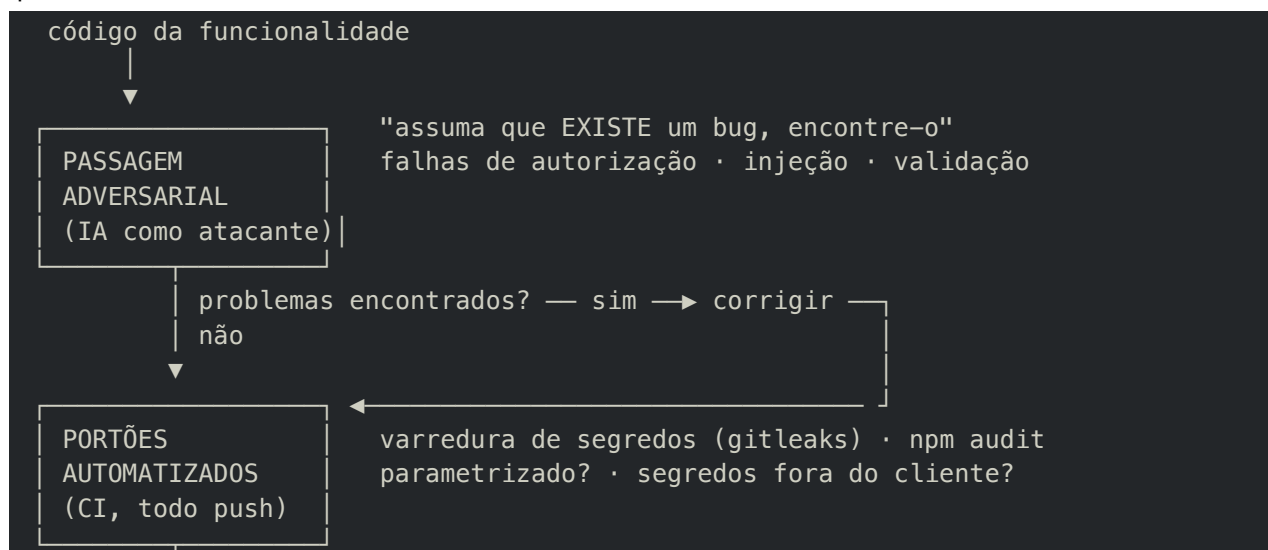
Para cada uma, mostre a requisição exata que explora isso, depois a correção.

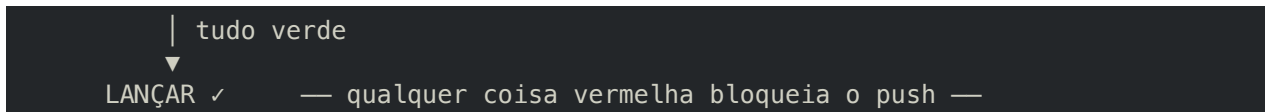
Não me tranquilize — assumo que EXISTE uma vulnerabilidade e a encontre.

Essa última linha importa: deixado neutro, a IA tende a dizer “parece seguro!”. Instruída a assumir que existe uma falha, ela de fato vai procurar. Combine a passagem adversarial com uma checklist curta de pré-lançamento que você roda em qualquer coisa voltada ao usuário:

- ☐ Todo endpoint verifica autorização, não apenas se o usuário está logado
- ☐ Todas as consultas ao banco de dados são parametrizadas — sem SQL montado por string
- ☐ A entrada do usuário renderizada na página passa por escaping (sem injeção de HTML bruto)
- ☐ Nenhum segredo no código do cliente, e nenhum commitado no repositório
- ☐ `.env` está fora do git; qualquer chave vazada já foi rotacionada
- ☐ Uploads de arquivo validam tipo e tamanho e usam nomes gerados
- ☐ Novas dependências foram checadas quanto à existência real e reputação

Imagine o portão como uma sequência que nada passa sem cumprir. A IA troca de chapéu de construtor para atacante, e depois as varreduras automatizadas cobrem o que uma passagem humana pode deixar passar:





E rode um **scanner de segredos** antes de fazer push — uma ferramenta como o `git leaks` (ou a varredura embutida da sua plataforma) vasculha seu código e histórico atrás de coisas com a forma de chaves. É uma rede de segurança de um único comando para o erro mais caro da lista, e você pode pedir para a IA integrá-la ao CI (Continuous Integration — o robô que roda suas verificações automaticamente a cada push de código) para que ela rode a cada push.

Complacência da automação, edição segurança

O momento mais perigoso não é o primeiro dia — é o dia noventa. No primeiro dia você é cauteloso; lê cada diff, roda o prompt de atacante, verifica o manipulador de upload. Aí cem mudanças são lançadas e nada de ruim acontece, e a verificação começa a parecer cerimônia. Você pula o portão de revisão “só dessa vez” numa mudança pequena. A mudança pequena adiciona um endpoint sem verificação de autorização. Nada acontece por um tempo, porque nada acontecer é exatamente a cara de um buraco de segurança visto de dentro.

Isso é complacência da automação apontada para a segurança, e é pior aqui do que em qualquer outro lugar, porque falhas de segurança são silenciosas e atrasadas. Uma funcionalidade quebrada te pune em minutos. Uma verificação de permissão quebrada te recompensa com sucesso aparente até o exato momento da violação. A ausência de desastre não é evidência de segurança — é a condição normal de um app vulnerável que simplesmente ainda não foi encontrado.

A defesa é tornar o portão barato o suficiente para você não pular e automático o suficiente para você não conseguir pular. Coloque a varredura de segredos e o `audit` no CI para que rodem sem depender da sua memória, e mantenha o prompt de atacante à mão para qualquer coisa que toque em autenticação, dados ou uploads. Segure uma linha: uma mudança que lida com dados de usuário ou permissões não é lançada até que alguém — você, com a ajuda honesta da IA — tenha tentado ativamente quebrá-la. Velocidade em vibe coding vem de bons portões, não de confiar que o silêncio significa que está tudo bem.

Recapitulação e prática

Principais lições

- Código gerado por IA é inseguro por padrão; assuma que existe uma vulnerabilidade e vá procurá-la em vez de confiar em uma primeira impressão limpa.
- As grandes categorias se repetem: injeção, segredos expostos, autorização ausente, uploads de arquivo inseguros e dependências inventadas.
- Autorização é por requisição, não por login — todo endpoint precisa verificar que este usuário pode fazer esta ação, não apenas que ele está logado.
- Rode um scanner de segredos como o `git leaks` antes de todo push, e rotacione qualquer chave que já tenha vazado; integre a varredura ao CI para que você não consiga pular.
- O dia noventa é mais perigoso que o primeiro dia — torne o portão de segurança barato o suficiente para não pular e automático o suficiente para você não conseguir pular.

Mão na massa

Pegue qualquer funcionalidade sua que toque em autenticação, dados de usuário ou uploads e rode o prompt de revisão adversarial abaixo contra ela. Para cada vulnerabilidade que a IA relatar, peça a requisição exata que a explora e a correção mínima, depois aplique a correção e rode o prompt de novo. Termine rodando o `git leaks` (ou a varredura de segredos da sua plataforma) sobre o repositório e o histórico.

Prompt do capítulo

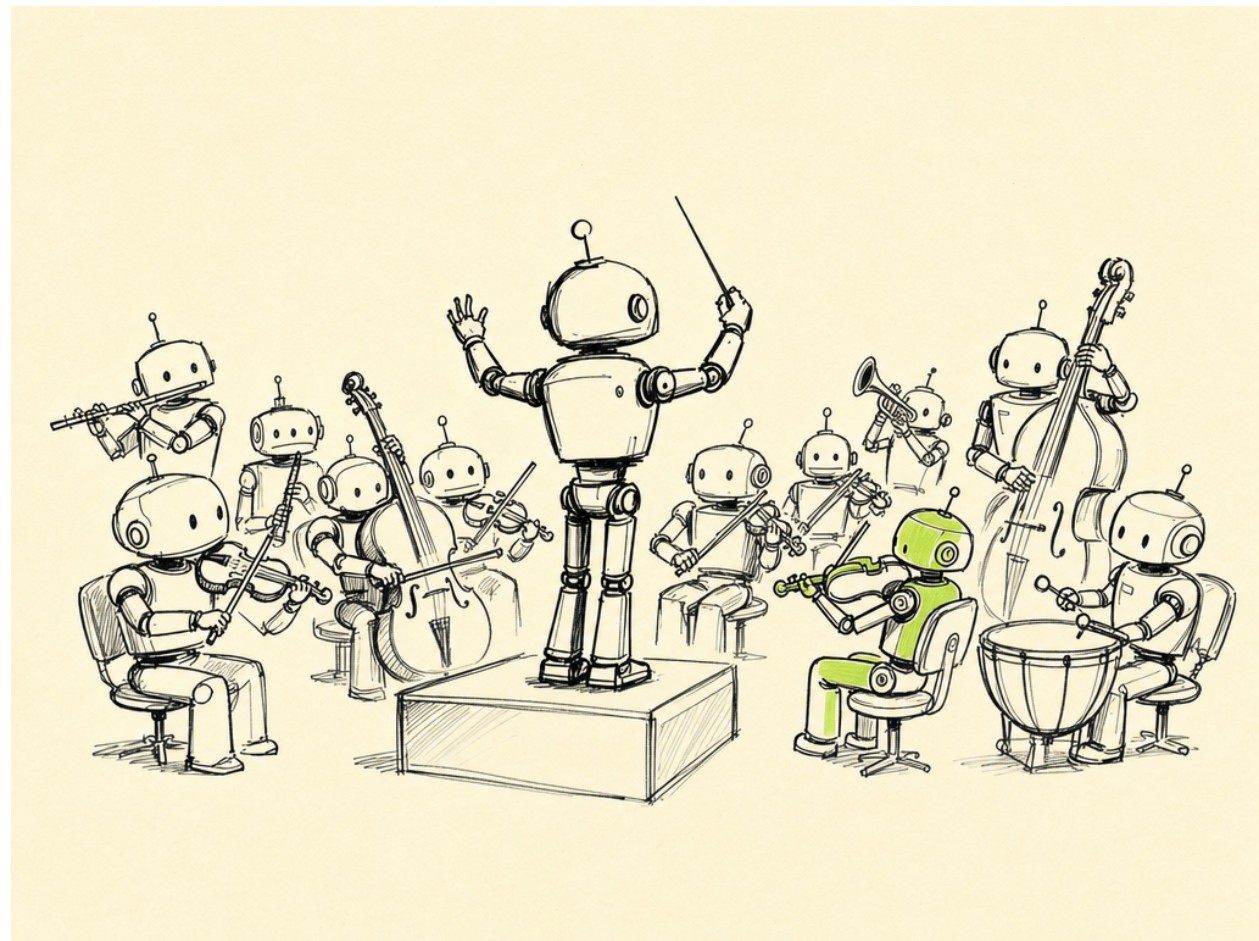
Aqui está o código de uma funcionalidade que lida com [autenticação / dados de usuário / upload] [cole o código].

Aja como um atacante revisando isso em busca de falhas de segurança. Verifique especificamente:

- autorização quebrada (um usuário pode agir sobre os dados de outro usuário?)
- injeção via entrada (injeção de SQL, XSS, injeção de comando)
- segredos ou chaves expostos no código ou nas respostas
- manipulação insegura de arquivos (tipo, tamanho, caminho, nomes gerados)
- validação ausente ou limites de taxa ausentes

Para CADA problema, mostre a requisição exata que o explora, depois a correção. NÃO me tranquilize – assumo que EXISTE uma vulnerabilidade e a encontre.

Orquestração de Agentes



Durante a maior parte deste livro, a IA foi uma ferramenta que você segura na mão: você pergunta, ela responde, você confere o resultado. Esse ainda é o modelo mental certo na maior parte do tempo. Mas as ferramentas ganharam um segundo modo — um em que você entrega uma *meta* em vez de um *passo*, e a IA trabalha por minutos (às vezes mais) antes de voltar até você.

Este capítulo é sobre essa mudança, e sobre não perder o fio da meada quando ela acontece. A tentação é ler “agente” como “agora posso parar de prestar atenção”. Essa é a leitura que coloca as pessoas em apuros. Um agente é alavancagem sobre um trabalho que você já entende — não uma licença para lançar

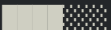
código que você parou de olhar.

As Três Camadas

Ajuda enxergar a assistência de IA como três camadas, cada uma te dando um pouco mais de corda.

- **Completions no editor.** A IA sugere a próxima linha ou bloco enquanto você digita. Você está no controle total; aceita ou rejeita cada sugestão. A unidade de trabalho é de poucas linhas, e você revisa todas elas por padrão porque aparecem bem abaixo do cursor.
- **Assistentes de nível de repositório.** Você conversa com um assistente que enxerga o projeto inteiro, edita vários arquivos de uma vez, roda comandos e lê a saída. É aqui que acontece a maior parte do trabalho descrito neste livro. A unidade de trabalho é um passo ou uma pequena funcionalidade, e você revisa o diff (a diferença entre o código antes e depois da mudança) após cada rodada.
- **Agentes autônomos.** Você dá uma meta — “faça a suíte de testes passar”, “migre este módulo para a nova API (Application Programming Interface, o conjunto de comandos que um software expõe para outros usarem)” — e o agente planeja, edita, roda, lê os erros e tenta de novo ao longo de muitos passos. Você revisa o *resultado*, não cada ação individual. A unidade de trabalho é a tarefa inteira.

Cada camada te dá mais corda: uma unidade de trabalho maior, menos direcionamento contínuo, e um ponto de revisão mais tardio:

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

O salto que importa é o último. Nas duas primeiras camadas você direciona continuamente; com um agente autônomo você o solta e confere o resultado. Essa é uma habilidade diferente, e é o assunto do resto deste capítulo.

Quando Deixar Rodar vs. Direcionar Manualmente

A decisão de deixar um agente rodar sem supervisão se resume a três perguntas — uma checklist que você percorre *antes* de sair de perto.

- **Raio de impacto.** Se algo der errado, quanto se quebra? Uma passada de formatação pelo código todo tem um raio de impacto pequeno. Uma migração de banco de dados ou uma mudança na lógica de autenticação tem um raio grande. Quanto maior o raio, mais você deve direcionar na mão.
- **Reversibilidade.** Dá para desfazer barato? Um trabalho feito em uma branch git nova, com a árvore de trabalho limpa, é reversível — `git restore .` e `some`. Um trabalho que apaga dados, envia e-mails, cobra cartões ou publica em produção não é. Nunca deixe um agente rodar sem supervisão sobre nada que você não possa desfazer.
- **Verificabilidade.** Dá para saber se o resultado está certo? Se “pronto” significa uma suíte de testes verde ou uma página que visivelmente funciona, você consegue verificar em segundos e um agente é uma ótima escolha. Se “pronto” depende de julgamento sutil — contas de dinheiro, regras de segurança, condições de corrida (race conditions) — você não consegue travar isso numa checagem rápida, então fica por perto.

Uma forma simples de combinar as três: **deixe rodar quando o trabalho for reversível e o resultado for barato de verificar; direcione na mão quando não for.** Traga a regra do capítulo sobre construir

funcionalidades — quanto maior o raio de impacto, menor o passo — e some a sua irmã: quanto maior o raio de impacto, mais curta a coleira.

Delimitando uma Tarefa Para Que Ela Não Se Perca

Um agente autônomo falha de uma maneira específica: ele se desvia. Você pede uma coisa, ele “gentilmente” refatora mais três, mexe em arquivos que você nunca mencionou, e volta com um diff que você não consegue nem ler. O conserto não é um agente mais inteligente — é um briefing mais rígido.

Uma boa tarefa autônoma tem quatro partes: uma **meta** clara, **restrições** explícitas, um **critério de pronto** concreto, e **guardrails** (barreiras de proteção) que o próprio agente consegue verificar. Os guardrails são a parte mais importante, porque são a cerca que o agente não consegue pular sem que você perceba.

Goal: Make the failing tests in tests/checkout/ pass.

Constraints:

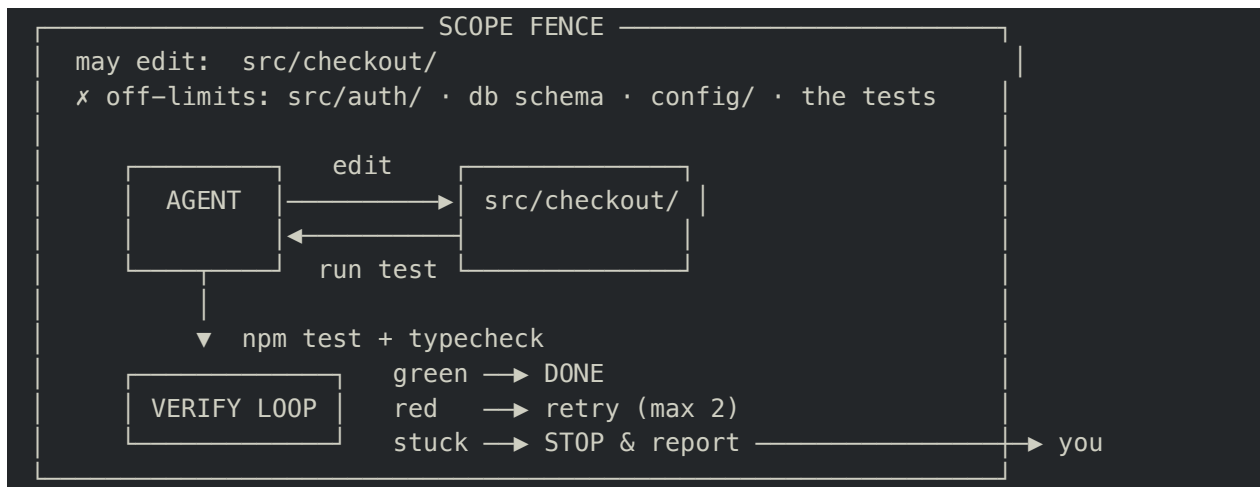
- Only edit files under src/checkout/. Do not touch src/auth/, the database schema, or anything in config/.
- Do not add new dependencies.
- Do not change the tests themselves — the tests define correct behavior.

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.

O briefing funciona porque encaixota o agente: as restrições isolam o que ele pode tocar, o critério de pronto é um comando que ele consegue rodar, e o laço de verificação o mantém honesto:



Três coisas tornam esse briefing difícil de desviar. As restrições desenham uma parede ao redor dos arquivos que o agente pode tocar. O critério de pronto é um comando que qualquer um consegue rodar, não uma sensação. E o último parágrafo dá permissão para *parar e perguntar* em vez de sair se debatendo — o que evita o desastre em câmera lenta em que um agente fica “consertando” coisas por vinte minutos e deixa a árvore pior do que encontrou.

Os testes são a cerca de verdade. Um agente que consegue rodar seus próprios testes tem um laço de retorno apertado e honesto e, na maior parte do tempo, fica dentro dele; um sem forma de checar seu

próprio trabalho vai te entregar, com toda a confiança, algo quebrado. Se você levar um único hábito deste capítulo: **dê ao agente uma forma de se verificar, e faça de passar nela a definição de pronto.**

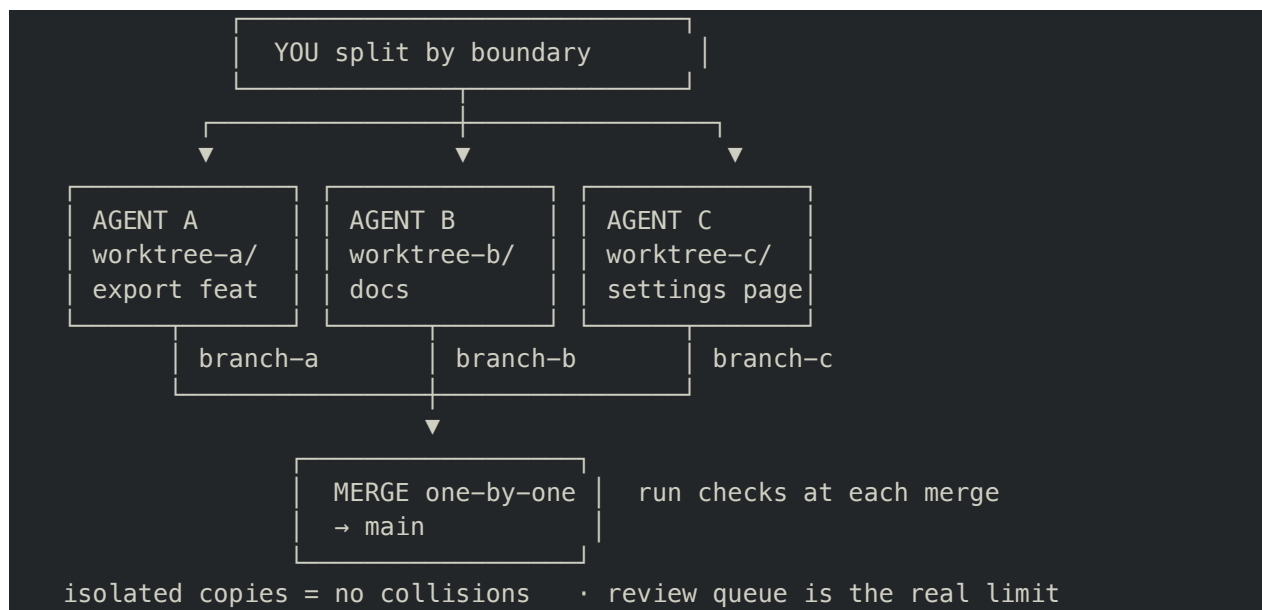
Agentes em Segundo Plano e em Paralelo

Depois que você confia em um agente numa tarefa delimitada, o próximo passo é óbvio: rodar vários ao mesmo tempo. Ferramentas modernas permitem disparar agentes em segundo plano — cada um na sua própria branch ou worktree (uma cópia de trabalho separada do projeto, para que as edições de um agente nunca colidam com as de outro) — de modo que três ou quatro tarefas delimitadas avancem enquanto você faz outra coisa.

Isso é um ganho real de velocidade, mas só para trabalho *independente*. O custo com o qual você está trocando é coordenação. Dois agentes editando o mesmo arquivo vão produzir diffs conflitantes, e resolver esse conflito na mão pode custar mais do que você economizou.

- **Divida por fronteira, não por número de linhas.** Dê a cada agente um pedaço que não se sobreponha aos outros — módulos separados, funcionalidades separadas, arquivos separados. “O Agente A faz a funcionalidade de exportação, o Agente B escreve a documentação” é limpo. “Os dois agentes editam `app.js`” é uma dor de cabeça de merge esperando para acontecer.
- **Isole os ambientes de trabalho.** Rode cada agente em sua própria branch ou worktree git, para que não pisem nas mudanças não commitadas um do outro. Faça o merge de volta um de cada vez, rodando suas checagens a cada merge.
- **Mantenha a contagem honesta.** Você ainda tem que revisar cada resultado. Quatro agentes rodando são quatro diffs para ler e quatro conjuntos de testes para confiar. Passando de um punhado, a fila de revisão — não os agentes — vira o seu gargalo.

Cada agente trabalha em sua própria worktree isolada, de modo que suas edições nunca colidem; você faz o merge de volta um de cada vez, rodando checagens a cada portão:



O erro é tratar o paralelismo como grátis. Não é — você transferiu o trabalho de *escrever* para *revisar e integrar*. Muitas vezes é uma ótima troca, mas só se você de fato fizer a revisão.

Multi-Agente: Fan-Out, Adversário, Síntese

Existe um segundo tipo de “muitos agentes” que roda *dentro* de uma única tarefa, em vez de entre tarefas separadas. Em vez de um agente fazer tudo, você divide os papéis:

- **Fan-out para amplitude.** Vários agentes atacam o mesmo problema de ângulos diferentes — ou exploram partes diferentes de um código grande — e depois reportam de volta. Útil quando você quer cobertura: “encontre todos os lugares onde chamamos essa função obsoleta” paraleliza bem.
- **Um revisor adversarial.** Um agente escreve; um segundo agente tem como única função *atacar* a saída do primeiro — achar o bug, o caso extremo esquecido, o erro não tratado. Um revisor apontado para um diff com a instrução “assuma que isso está errado e descubra por quê” pega coisas que o agente autor se convenceu de que estavam certas.
- **Síntese.** Uma passada final coleta os achados e produz um resultado coerente único — o código funcionando mais uma nota curta sobre o que o revisor sinalizou e como foi resolvido.

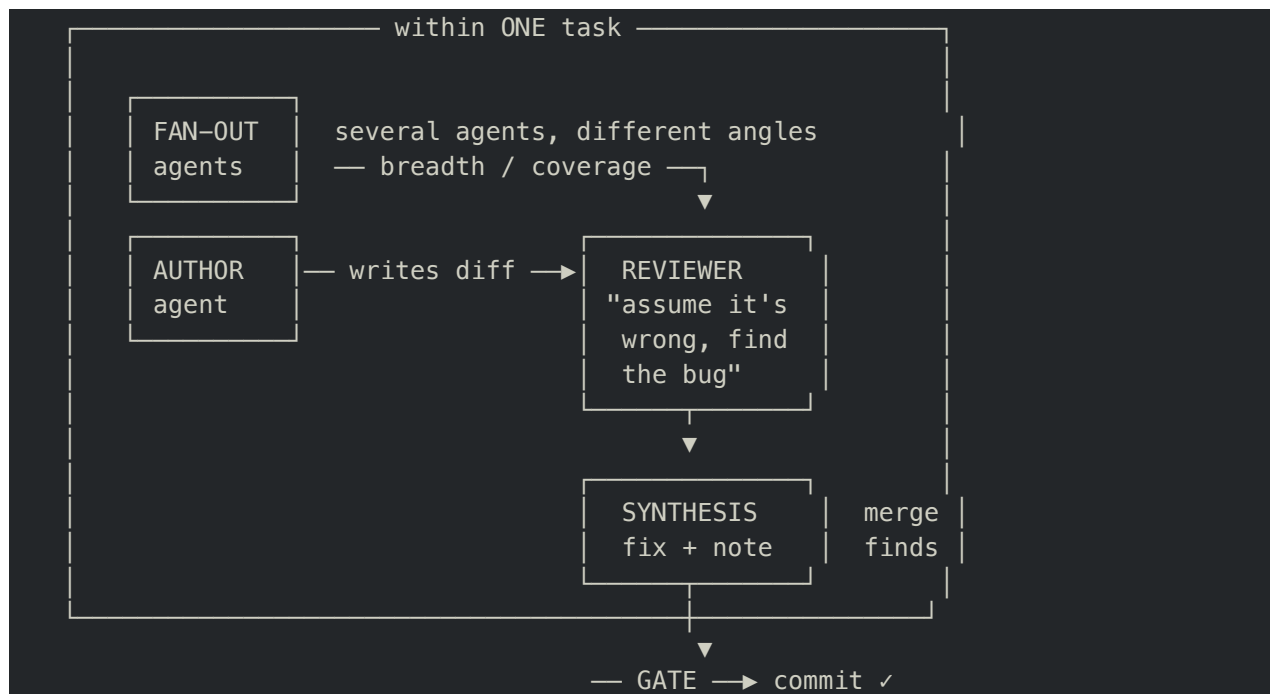
Reviewer agent brief:

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.

Os três papéis formam um pipeline com um portão no final: nada chega a um commit até o revisor adversarial ter feito sua passada e a síntese ter resolvido o que ele encontrou.



O revisor adversarial vale a pena montar mesmo quando você é uma pessoa só, sem uma frota de agentes — porque o autor de uma mudança é a pior pessoa para encontrar suas próprias falhas, seja humano ou IA. Uma segunda passada com um prompt hostil é um seguro barato.

Revisando Saídas em Escala

Aqui está o problema honesto: quando agentes estão escrevendo centenas de linhas em várias tarefas, você não consegue ler cada linha do jeito que revisaria um diff de dez linhas. Fingir que consegue leva ou a carimbar sem olhar (você aprova sem verificar) ou à paralisia (você nunca confia em nada). Nenhum dos

dois entrega resultado.

O caminho é travar em algumas camadas em vez de ler tudo:

- **Testes são o piso.** Nenhuma saída de agente entra em merge sem os testes relevantes verdes e o verificador de tipos limpo. Isso é inegociável e é automático — a máquina faz essa parte.
- **Revise pontualmente as partes de risco.** Você não consegue ler tudo, então leia as partes onde o raio de impacto é alto: qualquer coisa que toque autenticação, dinheiro, exclusão de dados ou chamadas externas. Passe os olhos no resto atrás dos sinais de alerta do capítulo sobre construir funcionalidades — arquivos que não deveriam ter sido tocados, coisas apagadas silenciosamente, um pedido pequeno que gerou um diff enorme.
- **Leia o diff, não o código inteiro.** `git diff --stat` conta em segundos o *quê* mudou e o quanto. Uma tarefa delimitada a um módulo que mostra mudanças em seis é um sinal para olhar com mais cuidado, antes mesmo de ler uma única linha.
- **Rode.** Uma suíte de testes verde não é um app funcionando. Clique na coisa. A checagem final é sempre a mesma de todos os outros capítulos: isso realmente faz o que você pediu, na sua frente?

A delimitação compensa duas vezes aqui. Uma tarefa que você delimitou de forma apertada produz um diff que você consegue de fato revisar; uma tarefa que você soltou produz um que você não consegue. O problema da revisão é, em grande parte, resolvido lá atrás, no briefing.

Custo e Loops Descontrolados

Um agente autônomo funciona em loop: tentar, checar, tentar de novo. Esse loop é a razão de ser dele — e também é o que pode, silenciosamente, estourar uma conta ou girar para sempre.

Isso se conecta com a ideia de segurança de custo de capítulos anteriores: **uma ferramenta por assinatura de preço fixo falha parando; um agente com API paga por uso falha cobrando**. Quando um assistente de código por assinatura atinge seu limite, ele simplesmente para de fazer mais nada até a janela resetar. Quando um script que você escreveu usando uma API paga trava num loop de repetição, ele continua chamando — e cada chamada custa dinheiro — até você perceber ou seu cartão estourar.

Então as regras são diretas:

- **Nunca rode um loop de agente sem limite contra uma API paga.** Se você escreveu seu próprio loop que chama uma API de LLM (Large Language Model, Modelo de Linguagem Grande — o tipo de IA por trás dessas ferramentas de código), ele precisa ter um teto rígido — um número máximo de iterações, um limite de gasto, um tempo limite. Sem exceções, sem “eu fico de olho”. Você não vai ficar.
- **Dê a cada loop uma saída.** “Pare depois de N tentativas”, “pare se os testes falharem duas vezes seguidas”, “pare se já estiver rodando há dez minutos”. Um agente sem uma condição de parada é um bug, não uma funcionalidade.
- **Observe as primeiras execuções.** Antes de confiar que um agente pode rodar em segundo plano, assista algumas execuções em primeiro plano do início ao fim. Você está checando se ele *converge* — se está chegando mais perto de terminar, e não travado girando em torno do mesmo conserto.
- **Prefira a ferramenta com guardrails.** Um assistente gerenciado com limites embutidos é mais seguro de deixar rodando do que um loop de API bruto que você escreveu à meia-noite. A assinatura falhando de forma fechada (parando) é uma vantagem; a API falhando de forma aberta (continuando a cobrar) é um risco.

Limites Honestos

Agentes são a forma mais poderosa de alavancagem deste livro, e é exatamente por isso que vale a pena ter clareza sobre o que eles são e o que não são.

Eles não são autonomia que você pode parar de entender. No momento em que você não consegue mais responder “o que esse código deveria fazer, e como eu saberia se está errado”, você cruzou de

usar alavancagem para torcer pela sorte. Cada camada deste capítulo — delimitação, guardrails, testes, revisão pontual, tetos de custo — existe para te manter do lado do entendimento dessa linha, enquanto deixa o agente fazer muito mais da digitação.

O teto de habilidade é o mesmo do resto do livro: sua capacidade de *dizer com precisão o que você quer e reconhecer quando você conseguiu*. Um agente multiplica essa capacidade; ele não a substitui. As pessoas que se queimam não são as que usaram agentes — são as que usaram agentes para parar de pensar. Não seja assim. Delegue o trabalho; mantenha o julgamento.

Recapitulação e Prática

Principais aprendizados

- Agentes multiplicam sua capacidade de dizer com precisão o que você quer e de reconhecer quando você conseguiu — eles não a substituem.
- A delimitação compensa duas vezes: uma tarefa bem delimitada produz um diff que você de fato consegue revisar, enquanto uma solta produz um que você não consegue.
- Revise o diff, não o código inteiro — `git diff --stat` sinaliza expansão indevida de escopo em segundos, e uma suíte de testes verde nunca substitui rodar a coisa.
- Nunca rode um loop de agente sem limite contra uma API paga; dê a cada loop um teto rígido — número máximo de iterações, limite de gasto ou tempo limite.
- Observe as primeiras execuções em primeiro plano para confirmar que o agente *converge* antes de confiar nele para rodar em segundo plano.

Mão na massa

Pegue uma tarefa pequena e bem delimitada e entregue-a a um agente com uma condição de parada explícita (“pare depois de 3 tentativas” ou “pare se os testes ainda falharem duas vezes”). Assista à execução do início ao fim em primeiro plano, depois revise apenas o diff com `git diff --stat` seguido de `git diff`. Se as mudanças tocarem mais arquivos do que você delimitou, esse é o seu sinal para apertar o briefing e rodar de novo.

Prompt do 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.
```