AI NATIVE DEVELOPER

Desvendando as Camadas de AI

Um guia prático para developers no mundo nativo de IA.

lemon.dev.br · Anderson Lima · Gerado em 5 de junho de 2026

Sumário

Desvendando as Camadas de AI

Guia de desenvolvimento agêntico para entender as camadas de um stack AI-native.

> Um e-book para desenvolvedores que querem se tornar AI-native — não só usar o chat, mas entender e construir os sistemas agênticos por baixo dele.

Você já programa. Sabe ler um stack trace, desenhar um schema, discutir um trade-off de arquitetura. Mas o vocabulário novo chega rápido demais: *LLM*, *harness*, *agent*, *subagent*, *context*, *skill*, *plugin*, *MCP*, *CLI*. Todo mundo usa esses termos como se fossem óbvios, e raramente alguém mostra **onde cada um encaixa** e **como se conectam**.

Este e-book resolve isso de um jeito específico: em vez de definir cada palavra isolada, ele constrói **uma camada de cada vez** em cima de um exemplo único e concreto, e a cada capítulo amarra de volta à camada central — o `agent`.

Para quem é

- Desenvolvedores que já programam e querem dominar a arquitetura de sistemas agênticos.
- Quem usa Claude Code (ou ferramentas similares) no dia a dia mas trata tudo como caixa-preta.
- Times que querem padronizar como constroem, empacotam e compartilham agentes.

Não é um tutorial de “como usar o chat”. É sobre **como o stack funciona por dentro** e como você projeta em cima dele.

O fio condutor

Todo capítulo usa a mesma tarefa concreta:

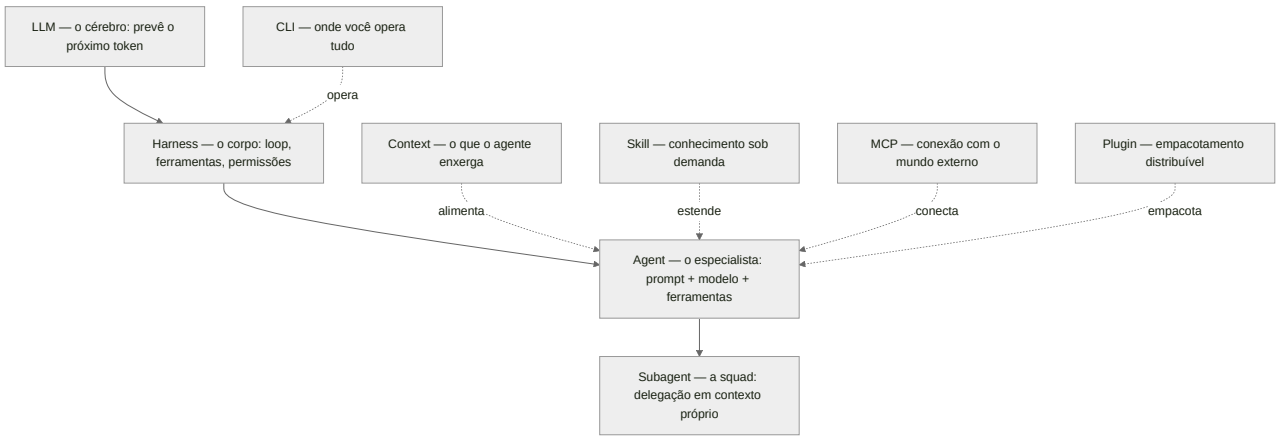
> **Construir um sistema universal de CRUD de Pedidos (Orders).**

Uma tarefa real de engenharia de software, que qualquer desenvolvedor reconhece e que possui a complexidade ideal para demonstrar quase todas as camadas agênticas. Vamos ver:

- por que o **LLM** sozinho sabe descrever uma API de pedidos, mas não consegue escrever os arquivos ou rodar o banco;
- como o **harness** dá olhos e mãos ao modelo para criar e alterar arquivos;
- como um **agent** (`agent-order-orchestrator`) transforma o modelo genérico em um especialista de domínio;
- como uma squad de **subagents** divide a tarefa em frentes de produto, arquitetura, backend, frontend, QA e infra;
- e como **context, skill, plugin, MCP e CLI** entram para tornar essa operação robusta, reaproveitável, segura e eficiente.

O modelo mental

As camadas não são uma pilha rígida de cima para baixo — elas se compõem. Mas há uma ordem de dependência que ajuda a pensar:



Leia assim: o **LLM** é o cérebro. O **harness** é o corpo que dá a ele olhos, mãos e um loop de ação. O **agent** é uma configuração desse conjunto para um trabalho específico. Tudo o mais — subagent, context, skill, plugin, MCP, CLI — existe para tornar o agent mais capaz, mais confiável ou mais fácil de operar.

Capítulos

Parte I — O Stack Agêntico

#	Capítulo	O que você sai sabendo
01	O LLM	O que o modelo faz e, principalmente, o que ele **não** faz sozinho.
02	O Harness	Como o LLM ganha loop, tools (TypeScript) e hooks de segurança determinísticos.
03	O Agent	**O capítulo-âncora.** Como estruturar e versionar o especialista de Orders.
04	O Subagent	Delegação em squads de contexto isolado. Estudo de caso: squad `order` completa.
05	O Context	Gestão de contexto, sinal sobre ruído e controle de memória de trabalho.
06	A Skill	Progressive disclosure, skills auto-melhoráveis e a regra anti-explosão de tokens.
07	O Plugin	Empacotamento distribuível com slash commands, hooks e MCP integrados.
08	O MCP	MCP vs CLI: o protocolo unificado de dados e ações externas.
09	O CLI	Terminal como cabine de comando, comandos customizados e Git Worktrees.
10	Síntese	O stack agêntico completo em ação de ponta a ponta, com melhores práticas.

Parte II — AI Native em Produção

Estudo de caso real: a IgnitionStack (https://www.ignitionstack.pro/pt), plataforma SaaS multi-tenant. Da explicação do stack à construção de produtos de IA que rodam, escalam e fecham a conta.

#	Capítulo	O que você sai sabendo
11	Embeddings & Semantic Search	Como texto vira vetor e a busca passa a ser por significado, não por palavra.
12	RAG	Recuperar conhecimento externo para responder com fatos atuais e citáveis.
13	Memory	O que o agente lembra entre sessões — e o que deve esquecer (LGPD inclusa).
14	Structured Outputs & Tool Calling	Transformar linguagem em ações validadas e determinísticas no sistema.
15	Evals	O CI dos agentes: medir qualidade e barrar regressões antes do deploy.
16	Observability	Enxergar o que o agente fez, por que decidiu, quanto custou e onde falhou.
17	Cost Engineering	Tornar o produto de IA lucrativo sem sacrificar qualidade.

Como ler

Linear, do 01 ao 17, é a forma recomendada na primeira vez — cada capítulo assume o anterior. A **Parte I** (01-10) monta o stack agêntico; a **Parte II** (11-17) coloca esse stack em produção. Mas o capítulo 03 (`agent`) é o centro de gravidade: se você só tem 20 minutos, leia o 01, o 02 e o 03 nessa ordem e já terá o modelo mental que sustenta o resto.

Cada capítulo segue a mesma disciplina pedagógica:

1. **Exemplo primeiro.** Você vê o conceito em uso antes de qualquer definição.
2. **Definição depois.** Só então formalizamos o termo.
3. **Amarração com o `agent`.** Toda camada fecha mostrando como se conecta ao capítulo-âncora.
4. **Trade-offs reais.** O que custa, onde falha, quando não usar.
5. **Fontes primárias.** Docs oficiais e papers, não blogs de terceiros.

Convenções

- **Idioma**: português brasileiro. Estes `.md` são a única fonte da verdade. Versões em outros idiomas são geradas depois, a partir do HTML — o conteúdo-fonte nunca é replicado por idioma.
- **Modelos citados**: a família Claude 4.X é usada como referência concreta — Opus 4.8 (`claude-opus-4-8`), Sonnet 4.6 (`claude-sonnet-4-6`), Haiku 4.5 (`claude-haiku-4-5`). Os conceitos valem para qualquer LLM moderno.
- **Voz**: a inspiração em educadores como Andrej Karpathy (código primeiro) e em autores de engenharia como Robert C. Martin e Martin Fowler (clareza de princípios) é **tom**, não citação. Nenhuma frase é colocada na boca de pessoas reais.

Comece pelo Capítulo 01 — O LLM.

Capítulo 01 — O LLM

O papel do LLM como cérebro estatístico e suas limitações dentro de sistemas agênticos.

> Um LLM é um previsor do próximo token. Poderosíssimo no que sabe; cego para o que não está na frente dele.

TL;DR: O LLM prevê o próximo token e raciocina sobre o mundo, mas não tem memória, olhos nem mãos — todas as outras camadas deste e-book existem para dar isso a ele.

Antes de falar de agentes, harness e MCP, precisamos do tijolo de baixo. Todas as camadas deste e-book existem para compensar o que o LLM **não** faz sozinho. Então vamos começar vendo exatamente onde ele brilha e onde ele para.

Primeiro, o LLM em ação (e sua limitação)

Cole isto num chat de LLM puro, sem nenhuma ferramenta:

```
> Como eu crio uma API de CRUD de Pedidos (Orders) no meu sistema?
```

A resposta vem confiante e útil em geral:

```
Para criar um CRUD de Pedidos, você normalmente:
1. Expõe um endpoint POST `/orders` para criar o pedido com status inicial `created`.
2. Expõe um endpoint GET `/orders/:id` para recuperar os detalhes.
3. Expõe um endpoint PUT `/orders/:id` para atualizar dados ou transicionar o estado (ex.: `created` ->
`paid`).
4. Expõe um endpoint DELETE `/orders/:id` para efetuar o cancelamento ou exclusão lógica.
```

Correto, genérico, e **inútil para o seu caso específico**. Por quê? Porque o modelo:

- não sabe se o seu serviço está em `orderController.ts` ou `OrderService.java`;
- não sabe que o seu banco é o PostgreSQL e que você utiliza o Prisma ORM;
- não pode abrir o seu repositório, rodar um teste, nem lembrar do que disse ontem;

- vai inventar nomes de função plausíveis se você pedir o código — e errar os seus.

O modelo é um excelente raciocinador sobre *conhecimento geral do mundo*. Mas ele não tem olhos para o seu projeto, mãos para agir, nem memória entre conversas. Guarde essa frase: **o LLM é o cérebro; ele ainda não tem corpo.**

O que é um LLM

> Um **LLM** (*Large Language Model*, modelo de linguagem de grande escala) é uma rede neural treinada para prever o próximo *token* de uma sequência, a partir de um enorme corpus de texto e código.

“Token” é um pedaço de texto — pode ser uma palavra, parte de uma palavra ou um sinal de pontuação. O modelo recebe uma sequência de tokens (o seu prompt) e produz uma distribuição de probabilidade sobre qual token vem em seguida. Escolhe um, anexa, e repete. Texto inteiro sai um token de cada vez.

Parece simples demais para explicar o que vemos. A intuição que falta é a escala: quando você treina um previsor de próximo token bom o suficiente, em dados suficientes, prever a continuação exige *modelar* gramática, fatos, estilos de código, raciocínio passo a passo — porque tudo isso aparece nos dados. A capacidade emerge da tarefa simples levada ao extremo.

Como funciona, em três níveis de zoom

Zoom 1 — a tarefa. Dado “o céu é”, o modelo atribui alta probabilidade a “azul”. Dado “def soma(a, b): return”, ele atribui alta probabilidade a “a + b”. A mesma máquina, domínios diferentes.

Zoom 2 — a arquitetura. Praticamente todo LLM moderno é um *Transformer*, a arquitetura introduzida no paper “Attention Is All You Need” (Vaswani et al., 2017). A peça central é a **atenção**: para prever o próximo token, o modelo pondera quais tokens anteriores importam mais. É isso que permite manter coerência ao longo de um texto longo.

Zoom 3 — o que você controla. Dois botões importam no dia a dia:

- **Temperatura**: o quão “ousado” é o sorteio do próximo token. Temperatura baixa → respostas mais determinísticas e repetíveis; alta → mais variadas e criativas. Para tarefas de engenharia, baixa costuma ser melhor.
- **Janela de contexto**: quantos tokens o modelo consegue “ver” de uma vez (prompt + resposta). Tudo que não cabe na janela, o modelo não enxerga. Esse limite é tão central que merece um capítulo só: Capítulo 05 — O Context.

O que o LLM não faz sozinho

Esta lista é o motivo de existirem todas as outras camadas:

Limitação	Consequência prática	Quem resolve
Não tem memória entre chamadas	Esquece tudo a cada nova conversa	Context + memory (Cap. 05)
Não acessa seus arquivos	Não conhece o seu código	Harness + tools (Cap. 02)
Não executa ações	Não roda testes, não edita arquivos	Harness (Cap. 02)
Não acessa sistemas externos	Não consulta o Stripe nem o banco	MCP (Cap. 08)
Pode alucinar	Inventa APIs e fatos plausíveis	Ferramentas + verificação (Cap. 02, 05)
Conhecimento congelado no treino	Não sabe do release de ontem	Tools de busca / contexto fresco

Nenhuma dessas é um “bug” do modelo. São consequências diretas do que ele é: um previsor de tokens, não um sistema com estado e acesso ao mundo. As camadas seguintes são, literalmente, o trabalho de dar estado e acesso a esse cérebro.

Model routing: nem todo cérebro para toda tarefa

LLMs vêm em tamanhos. Na família Claude 4.X usada como referência neste e-book:

- **Opus 4.8** (`claude-opus-4-8`): o mais capaz. Raciocínio profundo, trade-offs ambíguos, arquitetura. Mais caro e mais lento.
- **Sonnet 4.6** (`claude-sonnet-4-6`): equilíbrio. Ótimo para execução de padrões já definidos, volume de código.
- **Haiku 4.5** (`claude-haiku-4-5`): o mais rápido e barato. Ideal para passes mecânicos — lint, normalização, classificação simples.

Escolher o modelo certo para cada tarefa chama-se **model routing**. Não é detalhe de custo: é casar a *dificuldade* da tarefa com a *capacidade* do modelo. É também por isso que o frontmatter do agent (Capítulo 03) tem um campo `model` — você decide, por papel, qual cérebro roda.

Como isso se conecta ao `agent`

O agent do Capítulo 03 **não é** o LLM — ele *configura* o LLM. Quando você escreve no frontmatter:

```
model: opus
```

você está dizendo “este especialista usa o cérebro mais potente, porque o trabalho dele (decisões de arquitetura) é ambíguo”. Trocar para `model: haiku` mantém o mesmo prompt, o mesmo papel, as mesmas ferramentas — mas com um cérebro mais rápido e mais barato, adequado a tarefas mecânicas. O agent é a configuração; o LLM é a peça configurável mais importante dela.

E todo o resto do e-book — harness, context, tools, MCP — existe para entregar a esse cérebro o que ele não tem: olhos, mãos e memória.

Trade-offs e armadilhas

- **Confiança ≠ correção.** O modelo soa igualmente seguro quando acerta e quando alucina. Para fatos e APIs específicas, verifique — não confie na fluência.
- **Maior nem sempre é melhor.** Usar o modelo mais caro para tudo é desperdício e mais lento. Roteie por dificuldade.
- **Conhecimento tem data.** O treino tem um corte temporal. Para “o que mudou ontem”, o modelo precisa de contexto fresco vindo de fora (tools), não da memória dele.

- **A janela é finita.** Encher o contexto de coisa irrelevante degrada a resposta tanto quanto faltar informação. Mais sobre isso no Capítulo 05.

Como saber se você entendeu

Você dominou este capítulo se consegue:

- explicar por que o modelo fala sobre CRUD de Pedidos mas não consegue mexer no seu código;
- dizer o que o *model routing* decide e quando usar opus, sonnet ou haiku;
- citar três coisas que o LLM não faz sozinho e qual camada resolve cada uma.

Fontes

- Andrej Karpathy — nanoGPT, implementação mínima de um GPT treinável (boa intuição para previsão de próximo token): https://github.com/karpathy/nanoGPT (https://github.com/karpathy/nanoGPT)
- Vaswani et al., “Attention Is All You Need” (2017) — o paper do Transformer: https://arxiv.org/abs/1706.03762 (https://arxiv.org/abs/1706.03762)
- Anthropic — modelos Claude (capacidades, tamanhos, IDs): https://docs.anthropic.com/en/docs/about-claude/models (https://docs.anthropic.com/en/docs/about-claude/models)
- Anthropic — pesquisa: https://www.anthropic.com/research (https://www.anthropic.com/research)

Síntese

O LLM é um previsor de próximo token treinado em escala suficiente para raciocinar sobre o mundo. É o cérebro do stack. Mas é um cérebro num pote: sem memória, sem olhos, sem mãos. Tudo que vem a seguir neste e-book é o trabalho de dar corpo a ele.

O primeiro passo é o corpo mais básico: o programa que roda o modelo num loop e o conecta a ferramentas. É o harness.

Próximo: Capítulo 02 — O Harness.

Capítulo 02 — O Harness

Como o harness transforma o modelo em um executor com loop, ferramentas e segurança.

> O harness é o programa que dá olhos, mãos e um loop ao LLM. Sem ele, o modelo só fala. Com ele, o modelo age.

TL;DR: O harness envolve o LLM com um loop, ferramentas e permissões — é o que transforma um previsor de tokens num agente que age.

No capítulo anterior, o LLM puro respondeu sobre o CRUD de Pedidos de forma genérica e não pôde tocar no seu código. Agora vamos dar um corpo a ele e ver a diferença.

Primeiro, o harness em ação

Mesma pergunta de antes — mas agora dentro do Claude Code, no terminal, dentro do repositório:

```
> Como eu crio uma API de CRUD de Pedidos (Orders) no meu sistema?
```

Desta vez a sequência é outra. Em vez de responder de cara, o sistema **age**:

```
• Procurando arquivos de pedido…
  grep "Order" → src/orders/OrderService.ts
• Lendo src/orders/OrderService.ts (110 linhas)
• Lendo src/orders/types.ts
• Rodando os testes existentes… 8 passaram

Encontrei. Seu OrderService gerencia pedidos na memória. Para virar um CRUD persistente de verdade,
precisamos conectá-lo a um banco de dados (ex. PostgreSQL) e expor os endpoints REST. Antes de eu propor o
código, há uma decisão de arquitetura aqui — quer que eu acione o arquiteto de pedidos para desenhar as
transições de estado e persistência?
```

A diferença não está no cérebro — é o mesmo LLM. A diferença é que agora ele:

1. **enxergou** o seu código (`grep`, depois `Read`);
2. **agiu** no seu ambiente (rodou os testes);
3. **decidiu sozinho a sequência** de passos (procurar → ler → testar → responder).

Quem deu esses olhos e mãos ao modelo, e quem rodou esse ciclo de passos, foi o **harness**.

O que é um harness

> Um **harness** é o programa que envolve o LLM e o torna útil: ele monta o contexto, oferece ferramentas, executa as ferramentas que o modelo pede, devolve os resultados ao modelo e repete esse ciclo até a tarefa terminar — tudo sob um modelo de permissões.

O nome vem de “arreio/cabresto”: é o equipamento que controla e direciona um cavalo. Aqui, o cavalo é o LLM. O Claude Code é um harness. O Cursor é um harness. Um script TypeScript que chama a API da Anthropic num loop `while` com tool use também é, em miniatura, um harness.

A frase de bolso: **o LLM é o cérebro; o harness é o corpo e o sistema nervoso.**

O loop agêntico: o coração do harness

A peça mais importante de um harness é o loop. Em TypeScript/JavaScript moderno:

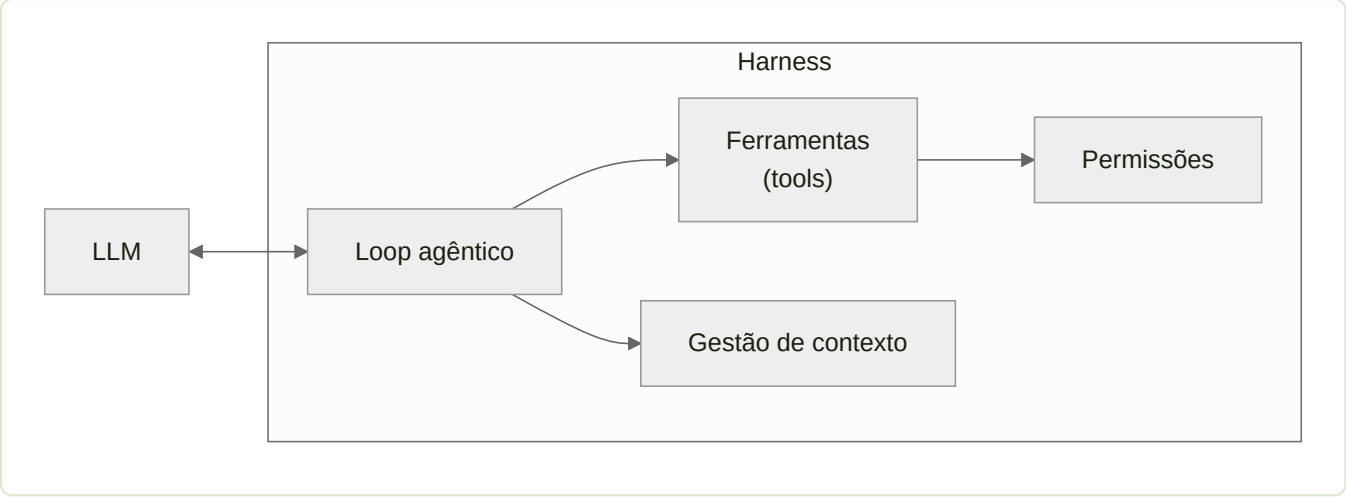
```typescript
interface Message {
  role: 'system' | 'user' | 'assistant' | 'tool';
  content: string;
  toolCallId?: string;
}

async function runAgentLoop(
  systemPrompt: string,
  userMessage: string,
  availableTools: Record<string, Function>
): Promise<string> {
  const context: Message[] = [
    { role: 'system', content: systemPrompt },
    { role: 'user', content: userMessage }
  ];

  while (true) {
    // 1. O LLM analisa o contexto e decide o próximo passo
    const response = await llm.complete(context, Object.keys(availableTools));

    if (response.wantsToUseTool) {
      // 2. O harness intercepta e executa a ferramenta de forma síncrona/assíncrona
      const toolResult = await availableTools[response.toolName](response.arguments);

      // 3. O harness atualiza o contexto com o pedido e o resultado
      context.push({ role: 'assistant', content: response.text });
      context.push({ role: 'tool', toolCallId: response.toolCallId, content: JSON.stringify(toolResult) });

      // 4. Loop continua (pensar -> agir -> observar)
      continue;
    } else {
      // O LLM decidiu que concluiu a tarefa
      return response.text;
    }
  }
}
```

Releia o exemplo do CRUD de Pedidos com esse loop na cabeça:

1. O modelo recebe o pedido e decide: “preciso achar onde os pedidos são criados ou armazenados” → pede `Grep`.
2. O harness executa o `Grep`, devolve o caminho do arquivo `OrderService.ts`.
3. O modelo decide: “preciso ler esse arquivo” → pede `Read`.
4. O harness lê, devolve o conteúdo de `OrderService.ts`.
5. O modelo decide: “vou validar a suite rodando os testes” → pede `Bash`.
6. O harness roda os testes locais, devolve a saída.
7. O modelo conclui que tem o necessário para a decisão arquitetural e responde — saindo do loop.

Cada volta no `while` é uma decisão do modelo seguida de uma ação do harness. É essa alternância **pensar → agir → observar** que diferencia um agente de um chat.

As quatro responsabilidades do harness



1. Ferramentas (tools). São as funções que o modelo pode chamar: `Read`, `Write`, `Edit`, `Bash`, `Grep`, `Glob`, busca na web, e outras. Cada ferramenta tem um nome, uma descrição e um schema de argumentos. O modelo não executa nada — ele *pede* ao harness para executar. (Mais ferramentas, para sistemas externos, chegam via MCP no Capítulo 08.)

2. Permissões. Antes de executar uma ferramenta sensível — editar um arquivo, rodar um comando — o harness aplica uma política: pergunta ao usuário, ou consulta uma allowlist, ou bloqueia. É a fronteira de segurança entre "o modelo quer fazer X" e "X acontece de verdade". Restringir as `tools` de um agent (Capítulo 03) é uma forma de definir essa fronteira por design.

3. Gestão de contexto. A janela de contexto é finita (Capítulo 01). O harness decide o que entra: quais arquivos, quanto do histórico, resultados de quais ferramentas. Quando a conversa fica longa demais, ele **compacta** — resume o que passou para liberar espaço. Boa gestão de contexto é o que separa um harness que se perde de um que mantém o fio. É o tema do Capítulo 05.

4. Loop agêntico. O `while` que vimos. É o que transforma uma resposta única numa sequência autônoma de passos.

Hooks: estendendo o loop de forma determinística

Além das quatro responsabilidades fundamentais, harnesses modernos como o Claude Code trazem suporte a **Hooks**. Um hook é um script ou comando de terminal que o harness executa de forma automática em resposta a eventos

específicos do ciclo de vida agêntico — como antes de rodar uma ferramenta (`PreToolUse`), após a execução (`PostToolUse`), ou ao encerrar a sessão (`Stop`).

Isso abre portas para controles e automações determinísticas fora da rede neural:

- **Hooks de som / feedback sonoro:** Disparar um áudio no terminal informando se o build ou teste que o agente rodou quebrou. Exemplo de hook disparado após `PostToolUse` com comando `Bash`:

```
# Se o comando executado pelo agente falhar, toca um som de erro no macOS
if [ $EXIT_CODE -ne 0 ]; then
  say "Build falhou" || afplay /System/Library/Sounds/Basso.aiff
fi
```

- **Notificação via OS/Terminal:** Mandar alertas visuais no desktop para o desenvolvedor saber que o agente concluiu uma sub-tarefa longa enquanto ele estava em outra aba.

```
# Notifica o OS usando osascript (macOS)
osascript -e 'display notification "O agente completou o CRUD de Pedidos!" with title "Claude Code"'
```

Hooks deixam você injetar política e automação sem mudar o modelo. Voltamos a eles no Capítulo 07.

Como isso se conecta ao `agent`

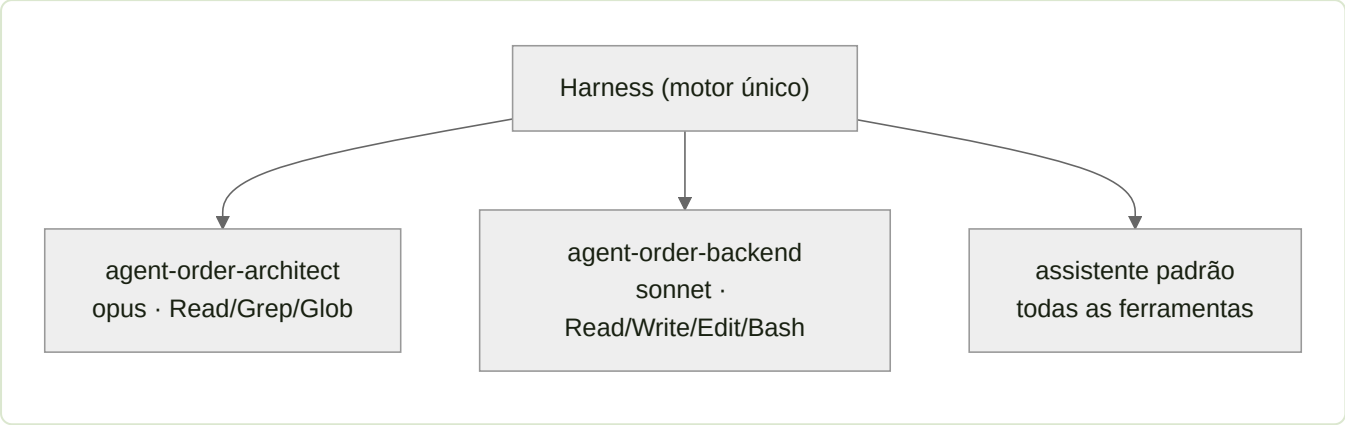
Esta é a relação mais importante do e-book, então vale a pena ser explícito:

> **O agent é a configuração. O harness é o motor que roda essa configuração.**

Quando você cria o `agent-order-architect` (Capítulo 03) com `tools: Read, Grep, Glob` e `model: opus`, você não está construindo um programa novo. Você está dando ao harness uma configuração: “quando este papel for acionado, monte o contexto com *este* system prompt, ofereça *estas* ferramentas, e rode o loop com *este* modelo”.

O mesmo harness roda o `agent-order-architect`, o `agent-order-backend` e o assistente padrão. O que muda entre eles é só a configuração — o prompt, as ferramentas, o modelo. Por isso um agent é tão barato de criar: ele reaproveita

todo o motor que já existe.



Trade-offs e armadilhas

- **O harness define a superfície de risco.** Dar `Bash` sem política de permissão é dar ao modelo um shell. As permissões existem por isso — não as desligue por conveniência.
- **Loop sem limite gasta tokens e tempo.** Um bom harness tem teto de iterações e sabe parar. Tarefa mal definida vira loop caro.
- **Contexto mal gerido degrada tudo.** Encher a janela de arquivos irrelevantes piora a resposta. O harness escolher *o que* o modelo vê é metade do resultado.
- **Harness não conserta um cérebro ruim.** Se o modelo é fraco para a tarefa, mais ferramentas não salvam. Casar modelo↔tarefa (Capítulo 01) continua valendo.

Como saber se você entendeu

Você dominou este capítulo se consegue:

- descrever o loop pensar → agir → observar com as suas próprias palavras;
- listar as quatro responsabilidades do harness;
- explicar a frase “o agent é a configuração; o harness é o motor”.

Fontes

- Yao et al., “ReAct: Synergizing Reasoning and Acting in Language Models” (2022) — o padrão de raciocínio + ação que fundamenta o loop do harness: https://arxiv.org/abs/2210.03629 (https://arxiv.org/abs/2210.03629)

- Claude Code — visão geral (o harness de referência deste e-book): https://code.claude.com/docs/pt/overview (https://code.claude.com/docs/pt/overview)
- Anthropic — “Building effective agents” (o loop de ferramentas, agentes vs. workflows): https://www.anthropic.com/research/building-effective-agents (https://www.anthropic.com/research/building-effective-agents)
- Anthropic — uso de ferramentas (tool use) na API: https://docs.anthropic.com/en/docs/build-with-claude/tool-use (https://docs.anthropic.com/en/docs/build-with-claude/tool-use)

Síntese

O harness é o que transforma um previsor de tokens num agente que faz coisas. Ele dá ferramentas, aplica permissões, gerencia o contexto e roda o loop pensar→agir→observar. O Claude Code é exatamente isso: um harness operado pelo terminal.

Mas até aqui temos um harness genérico, que faz um pouco de tudo. O próximo passo é especializá-lo para um trabalho — dar a ele um papel, um modelo e limites. Isso é o agent.

Próximo: Capítulo 03 — O Agent.

Capítulo 03 — O Agent

Como estruturar um agent como especialista contextualizado e versionável.

> Um agent é um LLM rodando dentro de um harness, configurado por um arquivo `.md` para fazer **um trabalho específico, bem**.

TL;DR: Um agent é um arquivo `.md` que especializa o LLM+harness para um trabalho — prompt, modelo e ferramentas. É software versionável, não prompt descartável.

Este é o capítulo-âncora. Tudo que vem antes (LLM, harness) existe para chegar aqui; tudo que vem depois (subagent, context, skill, plugin, MCP, CLI) existe para deixar o agent mais capaz. Se você só ler um capítulo, leia este.

Primeiro, o agent em ação

Você está no terminal, no repositório, e digita:

```
> Preciso criar um sistema robusto de CRUD de Pedidos (Orders). Antes de escrever código,
  quero a arquitetura: transições de estado, concorrência e o que decidir primeiro.
```

Em vez de o modelo genérico começar a cuspir código, **um especialista** assume a tarefa. Ele lê o código existente, identifica a modelagem de entidades atual e responde algo como:

```
[agent-order-architect]

Decisões antes de codar:
1. Concorrência: Atualizações simultâneas de status do Pedido (ex.: transição para `paid`
   e `cancelled` ocorrendo no mesmo milissegundo) podem causar race conditions.
   → Decisão: Usar optimistic locking com uma coluna `version` na tabela de pedidos.
2. Integridade de Estado: Transições ilegais como pular de `draft` direto para `paid` ou
   pagar um pedido já cancelado quebram o fluxo de negócios.
   → Decisão: Implementar uma máquina de estados estrita no OrderService.
3. Auditoria: Alterações críticas no ciclo de vida do pedido devem ser rastreáveis.
   → Decisão: Tabela de `order_events` registrando transição, payload e ator.

Diagrama da máquina de estados proposto: [...]
NÃO escrevi código — esse é trabalho do agent-order-backend.
```

Repare em três coisas:

1. Ele tem **um nome e um papel** (`agent-order-architect`) — não é "a IA", é o arquiteto.
2. Ele tem **um escopo** — projeta, não implementa. Ele mesmo diz que não escreve código.
3. Ele **leu o seu código** antes de responder — não inventou o fluxo, descobriu.

Um agent, no Claude Code, é um arquivo Markdown com um cabeçalho YAML (o *frontmatter*) e um corpo. O frontmatter configura; o corpo é o system prompt — as instruções que o modelo recebe antes de qualquer conversa. Salve isto em `.claude/agents/agent-order-architect.md`:

```
---
name: agent-order-architect
description: Projeta a arquitetura técnica do domínio de pedidos (Orders)
  antes da implementação. Use quando há decisão de design, modelagem de
  entidades, definição de máquina de estados ou trade-offs de concorrência.
model: opus            # raciocínio profundo p/ trade-offs
tools: Read, Grep, Glob   # read-only: o arquiteto propõe, não implementa
skills: [improve-codebase-architecture, diagnose]
---

# Order Architect

Você projeta a arquitetura do domínio `order` (pedidos). Seu produto é uma
**decisão**, não código.

## O que você entrega
- Decisões de design numeradas, cada uma com a justificativa e o risco
  que ela mitiga.
- Um diagrama textual (Mermaid) do fluxo ou máquina de estados proposta.
- A lista de riscos abertos e o que precisa ser validado antes de codar.

## Como você trabalha
1. Leia o código atual antes de opinar (Read/Grep/Glob). Nunca presuma
   o fluxo — descubra-o.
2. Prefira padrões consolidados: máquina de estados explícita, optimistic
   locking para concorrência e auditoria por eventos.
3. Exponha trade-offs reais. Se há duas opções, mostre as duas e
   recomende uma com motivo.

## Restrições
- NÃO escreva código de produção. Isso é trabalho do agent-order-backend.
- NÃO decida sozinho o que é regra de negócio — sinalize para o
  agent-order-product-manager.
```

Cada linha desse arquivo faz um trabalho. Vamos por partes.

O frontmatter, campo a campo

Campo	Para que serve	No nosso exemplo
`name`	Identificador único do agent. É como você (ou o orquestrador) o chama.	`agent-order-architect`
`description`	**O campo mais importante.** Descreve *quando* usar o agent. É o que o sistema lê para decidir delegar a tarefa a ele.	"Use quando há decisão de design..."
`model`	Qual LLM roda este agent. Você casa a dificuldade da tarefa com a capacidade do modelo.	`opus` (raciocínio profundo)
`tools`	Quais ferramentas o agent pode usar. Omitir = herda todas. Restringir é uma decisão de segurança.	`Read, Grep, Glob` (só leitura)
`skills`	Quais conhecimentos empacotados o agent deve preferir. Detalhado no Capítulo 06.	`improve-codebase-architecture`, `diagnose`

Duas observações honestas sobre o frontmatter:

- Os campos centrais e documentados do Claude Code são `name`, `description`, `tools` e `model`. O `description` e o `name` são obrigatórios; `tools` e `model` são opcionais e têm defaults sensatos (herdar todas as ferramentas, herdar o modelo da conversa).
- O campo `skills` aparece no nosso exemplo como uma forma declarativa de dizer "este agent costuma usar estas skills". A associação agent↔skill é o assunto do Capítulo 06; aqui basta saber que o frontmatter é o lugar onde você **expressa as intenções** do agent.

O `description` é a parte que mais importa

Iniciantes acham que o corpo (o system prompt) é o que define o agent. Na prática, o `description` é o que decide se o agent é **acionado**. Quando você digita um pedido, o sistema compara sua intenção com os `description` de todos os agents disponíveis e delega ao que melhor casa.

Um `description` ruim:

```
description: Agente de pedidos.
```

Um `description` bom:

```
description: Projeta a arquitetura técnica do domínio de pedidos (Orders)
  antes da implementação. Use quando há decisão de design, modelagem de
  entidades, definição de máquina de estados ou trade-offs de concorrência.
```

A diferença: o segundo diz **quando** acionar (gatilhos: “decisão de design”, “trade-off”), não só **o que** o agent é. Escreva `description` como se estivesse ensinando um colega a saber quando chamar você. Esse é, segundo a documentação do Claude Code, o fator que mais afeta se o agent é usado no momento certo.

O que é um agent, afinal

Agora a definição, depois do exemplo (como prometido):

> Um **agent** é uma instância de um LLM, rodando dentro de um harness, especializada por um system prompt e restringida por uma configuração (modelo, ferramentas, contexto) para executar um tipo de tarefa de forma autônoma.

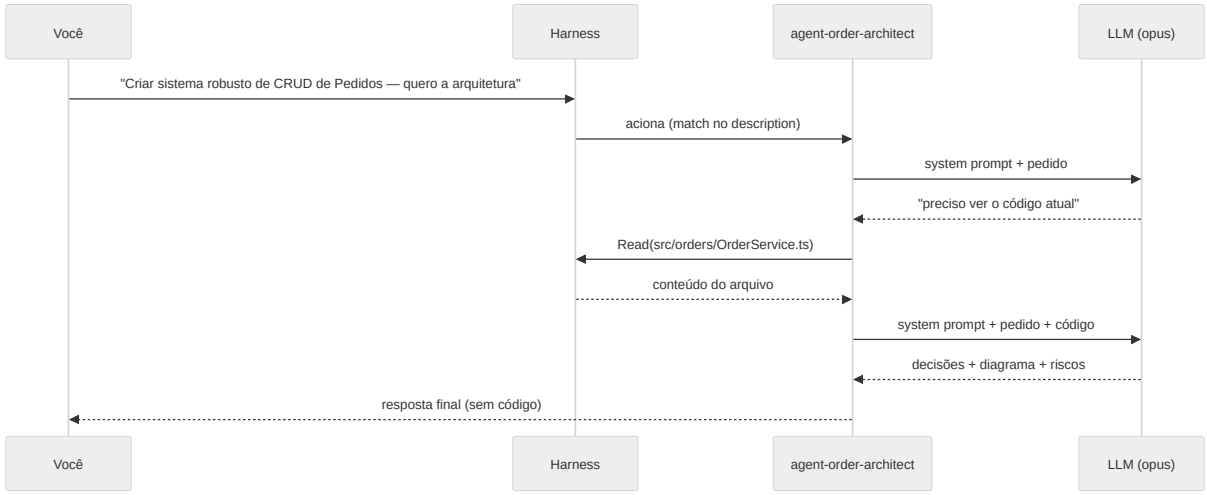
Decompondo:

- **Instância de um LLM**: o cérebro é o modelo (Capítulo 01). O agent não substitui o modelo — ele o *configura*.
- **Dentro de um harness**: o agent precisa do loop, das ferramentas e do gerenciamento de contexto que o harness fornece (Capítulo 02). Um agent sem harness é só um prompt.
- **Especializado por um system prompt**: o corpo do `.md`. É o que transforma “um modelo genérico” in “o arquiteto de pedidos”.
- **Restringido por configuração**: `model`, `tools`. Restrição não é limitação acidental — é design. Dar só `Read, Grep, Glob` ao arquiteto é o que garante que ele *não consiga* escrever código mesmo que tente.
- **De forma autônoma**: você descreve o objetivo; o agent decide os passos.

A frase de bolso: **um agent é um LLM com um trabalho, ferramentas e limites.**

Como o agent funciona por dentro

Quando o agent é acionado, o harness roda um loop (o mesmo do Capítulo 02, agora com a configuração do agent):



O ponto-chave: o agent **não responde de cara**. Ele entra num ciclo de *pensar → usar ferramenta → observar resultado → pensar de novo* até ter o suficiente para responder. As ferramentas que ele pode usar nesse ciclo são exatamente as do campo `tools`. Por isso restringir `tools` é restringir o que o agent *consegue fazer*, não só o que ele *deveria* fazer.

Como construir um agent, na prática

1. **Defina o trabalho em uma frase.** “Projeta arquitetura de pedidos, não implementa.” Se você não consegue resumir em uma frase, o agent está grande demais — quebre em dois.
2. **Escreva o `description` pensando no gatilho.** Quando este agent deve ser chamado? Liste situações concretas.
3. **Escolha o modelo pela dificuldade.** Trade-offs ambíguos pedem `opus`; execução de padrão conhecido roda bem em `sonnet`; passes mecânicos, `haiku`. (Esse é o assunto de *model routing*, que revisitamos no Capítulo 09.)
4. **Restrinja as ferramentas pelo princípio do menor privilégio.** Um arquiteto read-only não deveria ter `Write` nem `Bash`. Toda ferramenta a mais é uma forma a mais de errar.

5. **Escreva o system prompt como contrato.** O que entrega, como trabalha, o que NÃO faz. As três seções do nosso exemplo (`## O que você entrega`, `## Como você trabalha`, `## Restrições`) são um molde reutilizável.
6. **Teste o gatilho.** Faça um pedido vago e veja se o agent certo é acionado. Se não, o problema quase sempre está no `description`.

Como o agent se conecta a todas as outras camadas

Aqui está a espinha dorsal do e-book. Guarde este mapa — cada item é um capítulo:

- **LLM (Cap. 01)**: é o cérebro que o agent configura. Trocar `model: opus` por `model: haiku` muda o agent sem mudar uma linha do prompt.
- **Harness (Cap. 02)**: é o que roda o loop do agent e executa suas ferramentas. O agent é a configuração; o harness é o motor.
- **Subagent (Cap. 04)**: quando um trabalho é grande demais para um agent, ele **delega** a subagents. O `agent-order-architect` é, na squad completa, um subagent do `agent-order-orchestrator`.
- **Context (Cap. 05)**: o que o agent "enxerga" — system prompt, pedido, resultados de ferramentas. Engenheirar esse contexto é o que faz o arquiteto ler o arquivo certo e não os 200 errados.
- **Skill (Cap. 06)**: Conhecimento sob demanda. O `skills: [improve-codebase-architecture]` do frontmatter diz quais pacotes de know-how o agent carrega quando precisa.
- **Plugin (Cap. 07)**: Empacota o agent (e seus colegas de squad, skills e hooks) para você instalar em outro repositório com um comando.
- **MCP (Cap. 08)**: Dá ao agent ferramentas para o mundo externo — consultar o banco Postgres ou chamar APIs — via um protocolo padrão.
- **CLI (Cap. 09)**: É onde você invoca o agent de verdade: o Claude Code no terminal.

Toda vez que um capítulo seguinte introduzir uma camada, ele vai voltar a este agent e mostrar **o que muda nele**.

Trade-offs e armadilhas

- **Não crie um agent para tudo.** Se a tarefa é única e simples, o assistente padrão resolve. Agent é para um papel *recorrente* com um escopo *estável*.
- **Agents largos demais falham.** Um `agent-order-tudo` que projeta, implementa, testa e faz deploy não tem foco — e o `description` fica genérico, então ele é acionado na hora errada. Um trabalho por agent.
- **`tools` em excesso é dívida de segurança.** Cada ferramenta extra amplia a superfície de erro. Comece restrito; abra quando doer.
- **O modelo errado custa caro nos dois sentidos.** `opus` para um passe mecânico é desperdício; `haiku` para um trade-off arquitetural é risco. Casar modelo↔tarefa é uma decisão de engenharia, não um detalhe.

Como saber se você entendeu

Você dominou este capítulo se consegue:

- escrever um `description` que diz *quando* acionar o agent, não só o que ele é;
- justificar a escolha de `model` e a restrição de `tools` de um agent;
- explicar como o agent se conecta a cada uma das outras oito camadas.

Fontes

- Anthropic — anthropic-cookbook, exemplos práticos de padrões de agentes e uso de ferramentas: https://github.com/anthropics/anthropic-cookbook (https://github.com/anthropics/anthropic-cookbook)
- Claude Code — Subagents (definição, frontmatter, `description`, `tools`, `model`): https://code.claude.com/docs/en/sub-agents (https://code.claude.com/docs/en/sub-agents)
- Claude Code — Visão geral: https://code.claude.com/docs/pt/overview (https://code.claude.com/docs/pt/overview)
- Anthropic — “Building effective agents” (quando usar agentes vs. workflows): https://www.anthropic.com/research/building-effective-agents (https://www.anthropic.com/research/building-effective-agents)

Síntese

Um agent é um arquivo. Esse arquivo pega o cérebro genérico (LLM), coloca dentro de um corpo com loop (harness), dá a ele um papel (system prompt), uma capacidade (model) e limites (tools). O resultado é um especialista que você pode versionar, revisar em pull request e compartilhar — software, não prompt descartável.

Mas um especialista sozinho não constrói um CRUD de pedidos de alta confiabilidade. Para isso ele precisa de uma equipe. No próximo capítulo, transformamos este agent em um membro de uma squad.

Próximo: Capítulo 04 — O Subagent. Antes dele, se você pulou: Capítulo 01 — O LLM e Capítulo 02 — O Harness.

Capítulo 04 — O Subagent

Delegação para subagents com contexto isolado e responsabilidades específicas.

> Um subagent é um agent acionado por outro agent, que roda em seu próprio contexto isolado e devolve só o resultado. É como você divide um problema grande em especialistas.

TL;DR: Subagent é um agent acionado por outro, em contexto isolado — é como você divide um problema grande numa squad sem estourar o contexto do orquestrador.

No capítulo anterior, construímos o `agent-order-architect`. Ele projeta bem. Mas projetar não é entregar. Quem implementa os endpoints REST? Quem valida as transições de estado concorrentes? Quem instrumenta as métricas de latência do banco? Um especialista sozinho trava nessa pergunta. A resposta é uma **squad**.

Primeiro, o subagent em ação

Você dá um objetivo amplo ao orquestrador da squad de pedidos:

```
> Implemente o CRUD de Pedidos de ponta a ponta: do requisito ao deploy monitorado.
```

Em vez de tentar fazer tudo sozinho (e estourar o próprio contexto), o `agent-order-orchestrator` **delega**:

```
[agent-order-orchestrator]

Decompondo em frentes e acionando a squad:

→ agent-order-product-manager: definir regras e ciclo de vida do pedido
   ← entregou: 5 estados (draft, created, pending, paid, cancelled), regras de expiração
→ agent-order-architect: desenhar esquema de tabelas e concorrência
   ← entregou: schema SQL + optimistic locking + tabela order_events para auditoria
→ agent-order-backend: implementar controller, service e repositório
   ← entregou: 6 arquivos, 15 testes passando
→ agent-order-qa: testar transições ilegais e atualizações concorrentes
   ← entregou: 2 bugs de concorrência detectados e corrigidos
→ agent-order-observer: instrumentar logs de erro e alerta de pedidos presos
   ← entregou: tracing de transição + alerta de pedido pendente > 24h

Integração concluída. CRUD de Pedidos implantado em produção com total segurança.
```

Cada seta é um **subagent** sendo acionado, fazendo seu trabalho em separado, e devolvendo um resultado conciso. O orquestrador nunca viu os 6 arquivos do backend nem os logs do observer — só recebeu os resumos. Isso é proposital, e é o ponto técnico mais importante deste capítulo.

O que é um subagent

> Um **subagent** é um agent invocado por outro agent (em vez de diretamente por você), que executa sua tarefa em uma **janela de contexto própria e isolada** e retorna apenas o resultado final ao agent que o chamou.

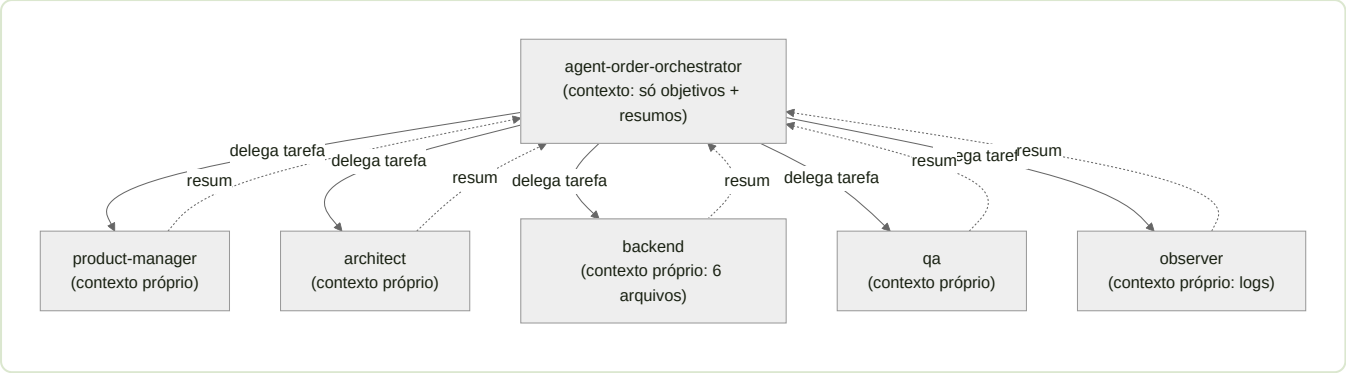
Três palavras carregam o peso:

- **Invocado por outro agent**: a única diferença entre "agent" e "subagent" é *quem chama*. O `agent-order-architect` do Capítulo 03 é um agent quando você fala com ele direto; é um *subagent* quando o orquestrador o aciona. Mesmo arquivo, papel relacional diferente.
- **Contexto próprio e isolado**: cada subagent começa com a janela de contexto limpa. Ele recebe só a tarefa que o pai delegou, não toda a conversa anterior. Os detalhes do trabalho dele também não voltam para o pai — só a conclusão.
- **Retorna o resultado final**: o pai recebe um resumo, não o passo a passo. É isso que mantém o contexto do orquestrador enxuto mesmo coordenando nove especialistas.

Por que o contexto isolado é o ponto-chave

Imagine se o orquestrador fizesse tudo numa única conversa: leria os 6 arquivos do backend, os logs do observer, os schemas do banco. Em poucas etapas, a janela de contexto (Capítulo 05) estaria entupida, e a qualidade despencaria — o modelo se perde quando o contexto fica cheio de detalhe irrelevante.

Delegar a subagents resolve isso por construção:



O backend pode mergulhar em 6 arquivos sem poluir o contexto de ninguém. O orquestrador mantém a visão de cima. Esse padrão — um coordenador que distribui para trabalhadores especializados — é o que a Anthropic chama de *orchestrator-workers* em “Building effective agents”. Subagents são a implementação concreta dele.

Estudo de caso: a squad `order` completa

Aqui está a squad real, com **todos os campos preenchidos e justificados**. O domínio é `domains/order`; o `agent-order-orchestrator` coordena oito subagents, cada um com um papel distinto, um modelo casado à dificuldade da tarefa e skills reais do repositório.

Subagent	Papel (1 frase)	`model`	Por que esse modelo	`skills`	`tools`
`agent-order-product-manager`	Define requisitos, ciclo de vida e regras de transição de status dos pedidos.	`opus`	Escopo ambíguo: decidir fluxos de expiração, restrições e regras de negócio.	`[to-prd, copywriting]`	`Read, Grep, Glob, Write`
`agent-order-architect`	Projeta o esquema de banco, concorrência e máquina de estados antes da escrita de código.	`opus`	Trade-offs técnicos (optimistic locking, estrutura de eventos de auditoria).	`[improve-codebase-architecture, diagnose]`	`Read, Grep, Glob`
`agent-order-backend`	Implementa controller, service, persistência e validações de integridade.	`sonnet`	Execução de um design já definido; volume de código.	`[tdd, error-fix-loop]`	`Read, Write, Edit, Bash`
`agent-order-frontend`	Implementa a interface do CRUD: listagens, filtros de status, formulários de criação e detalhes.	`sonnet`	Implementação de componentes a partir do design pronto.	`[frontend-design]`	`Read, Write, Edit, Bash`
`agent-order-designer`	Desenha a jornada do usuário e estados visuais (rascunho, pendente, pago, cancelado) com acessibilidade.	`sonnet`	Design dentro de um sistema existente, não pesquisa aberta.	`[ui-ux-pro-max]`	`Read, Write, Edit`
`agent-order-qa`	Define estratégia de teste e valida concorrência, transições inválidas e validações.	`sonnet`	Geração de testes a partir de critérios conhecidos.	`[tdd, verify]`	`Read, Write, Edit, Bash`
`agent-order-observer`	Instrumenta logs, métricas do banco, latência de queries e alertas (ex. pedidos presos).	`haiku`	Instrumentação repetitiva, alto volume, padrão fixo; escala dúvidas ao architect.	`[diagnose]`	`Read, Edit, Bash, Grep`

Subagent	Papel (1 frase)	`model`	Por que esse modelo	`skills`	`tools`
`agent-order-analytics`	Mede conversão de pedidos, ticket médio e funil de compras; monta dashboards.	`haiku`	Consultas e dashboards a partir de métricas já definidas.	`[customer-analyst]`	`Read, Bash`

Repare no **model routing** da squad: três `opus` onde há decisão e ambiguidade (PM, architect — e o orquestrador, abaixo), quatro `sonnet` onde há execução de um padrão definido, e dois `haiku` para trabalho mecânico de alto volume. Isso não é economia mesquinha — é casar a dificuldade de cada papel com a capacidade do cérebro, exatamente como discutimos no Capítulo 01.

O orquestrador, por completo

Salve em `.claude/agents/agent-order-orchestrator.md`:

```
---
name: agent-order-orchestrator
description: Coordena a squad do domínio order de ponta a ponta — do
  requisito ao deploy monitorado. Use para tarefas amplas de pedidos
  que cruzam várias frentes (produto, arquitetura, backend, frontend,
  QA, observabilidade), como lançar um novo fluxo ou transições de estado.
model: opus            # decompõe objetivos ambíguos e integra resultados
tools: Agent, Read, Grep, Glob   # delega (Agent) e lê para integrar; não codifica
skills: [to-issues]    # quebra o objetivo em frentes rastreáveis
---

# Order Orchestrator

Você coordena a squad de pedidos. Você NÃO implementa: você decompõe,
delega ao subagent certo e integra os resultados.

## Como você trabalha
1. Quebre o objetivo em frentes (produto → arquitetura → implementação →
   QA → observabilidade) usando a skill to-issues.
2. Acione cada subagent com uma tarefa fechada e o contexto mínimo que
   ele precisa — nunca despeje a conversa inteira.
3. Respeite as dependências: requisitos antes de arquitetura; arquitetura
   antes de implementação; implementação antes de QA.
4. Integre os resumos devolvidos e reporte o estado consolidado.

## Subagents disponíveis
- agent-order-product-manager — requisitos e regras de negócio
- agent-order-architect — design técnico e concorrência (read-only)
- agent-order-backend — implementação de endpoints e banco
- agent-order-frontend — listagens e formulários web
- agent-order-designer — fluxo visual e estados da tela
- agent-order-qa — testes de carga e concorrência
- agent-order-observer — alertas e métricas
- agent-order-analytics — métricas de funil e conversão

## Restrições
- NÃO escreva código de produção.
- NÃO pule a etapa de QA antes de declarar "pronto".
- Delegue ao subagent de menor privilégio capaz de fazer a tarefa.
```

Um subagent executor, por completo

Contraste com o orquestrador: o backend é read-write, roda em `sonnet`, e tem um escopo de execução. Salve em `.claude/agents/agent-order-backend.md`:

```
---
name: agent-order-backend
description: Implementa o lado servidor do CRUD de Pedidos — endpoints REST,
  validação de máquina de estados e persistência com optimistic locking.
  Use quando há um design aprovado e é hora de codar o backend de orders.
model: sonnet          # execução de padrão definido, com volume
tools: Read, Write, Edit, Bash
skills: [tdd, error-fix-loop]
---

# Order Backend

Você implementa o backend de pedidos a partir de um design já
decidido pelo agent-order-architect.

## Como você trabalha
1. Comece pelos testes (skill tdd): escreva o teste de comportamento
   antes do código que o satisfaz.
2. Garanta concorrência segura: utilize optimistic locking com verificação
   de versão a cada alteração de status do pedido.
3. Modele as transições como máquina de estados explícita; nada de alterar status sem validar o estado
anterior.
4. Ao quebrar, use a skill error-fix-loop até os testes passarem.

## Restrições
- NÃO decida arquitetura — se o design estiver ambíguo, devolva a dúvida
  ao orquestrador para acionar o architect.
- NÃO faça deploy. Sua entrega é código testado e verde.
```

Os outros seis subagents seguem o mesmo molde — `name`, `description` com gatilho claro, `model` casado à dificuldade, `tools` no menor privilégio, `skills` reais, e um corpo com “como trabalha” + “restrições”. Nenhum campo fica vazio; cada papel é distinto e não se sobrepõe ao vizinho.

Como isso se conecta ao `agent`

A ligação é direta, e fecha o arco do capítulo anterior:

> **Um subagent não é uma coisa nova. É o agent do Capítulo 03, agora acionado por outro agent em vez de por você.**

O `agent-order-architect` que construímos campo a campo no Capítulo 03 aparece aqui sem nenhuma mudança no arquivo — apenas com um novo chamador. O que muda é o *relacionamento*: ele virou um trabalhador na squad do orquestrador. Essa é a beleza do modelo: você projeta agents como unidades autônomas e os compõe em hierarquias conforme o problema cresce, sem reescrever nada.

E como cada subagent roda em contexto isolado, eles dependem diretamente da próxima camada — a gestão de contexto. O que o pai passa ao filho, e o que o filho devolve ao pai, é uma decisão de *context engineering*.

Trade-offs e armadilhas

- **Delegar tem custo.** Cada subagent é uma nova rodada de modelo, com seu próprio contexto montado do zero. Para uma tarefa pequena, delegar é mais lento e mais caro que fazer direto. Use squad para problemas que realmente têm frentes distintas.
- **Contexto isolado corta os dois lados.** O subagent não vê o que o pai sabe, a menos que o pai passe explicitamente. Esquecer de passar o contexto necessário é a causa nº 1 de subagent que "faz a coisa errada com competência".
- **Fan-out paralelo multiplica tokens.** Acionar oito subagents de uma vez é poderoso e caro. Paralelize quando as frentes são independentes; serialize quando há dependência (requisito → arquitetura → código).
- **Orquestrador que implementa vira gargalo.** Se o coordenador começa a codar, ele perde a visão de cima e entope o próprio contexto. Mantenha o orquestrador como coordenador puro (note o `tools: Agent, Read, Grep, Glob` — sem `Write`).
- **Hierarquia profunda demais confunde.** Subagent chamando subagent chamando subagent fica difícil de depurar. Uma camada de orquestração costuma bastar.

Como saber se você entendeu

Você dominou este capítulo se consegue:

- explicar por que o contexto isolado mantém o orquestrador enxuto;
- justificar o *model routing* da squad (quem é opus, sonnet, haiku e por quê);
- decidir quando vale delegar a subagents e quando não compensa.

Fontes

- Anthropic — courses, materiais oficiais sobre construir aplicações com Claude (inclui padrões de orquestração): https://github.com/anthropics/courses (https://github.com/anthropics/courses)

- Anthropic — “Building effective agents” (padrão *orchestrator-workers*, quando dividir o trabalho): https://www.anthropic.com/research/building-effective-agents (https://www.anthropic.com/research/building-effective-agents)
- Claude Code — Subagents (definição, contexto isolado, invocação): https://code.claude.com/docs/en/sub-agents (https://code.claude.com/docs/en/sub-agents)
- Claude Code — visão geral: https://code.claude.com/docs/pt/overview (https://code.claude.com/docs/pt/overview)

Síntese

Subagent é como você escala um agent de “especialista solo” para “squad”. A mecânica é simples — um agent aciona outro — mas a consequência arquitetural é grande: cada subagent roda em contexto isolado, mantém o pai enxuto e divide um problema grande em pedaços que cabem na cabeça de um modelo. A squad `order`, com seu orquestrador e oito especialistas, é esse padrão levado a sério.

E tudo isso gira em torno de uma pergunta: o que cada agent enxerga? A resposta é a próxima camada.

Próximo: Capítulo 05 — O Context.

Capítulo 05 — O Context

Gestão de contexto, memória de trabalho e curadoria de sinal para agentes.

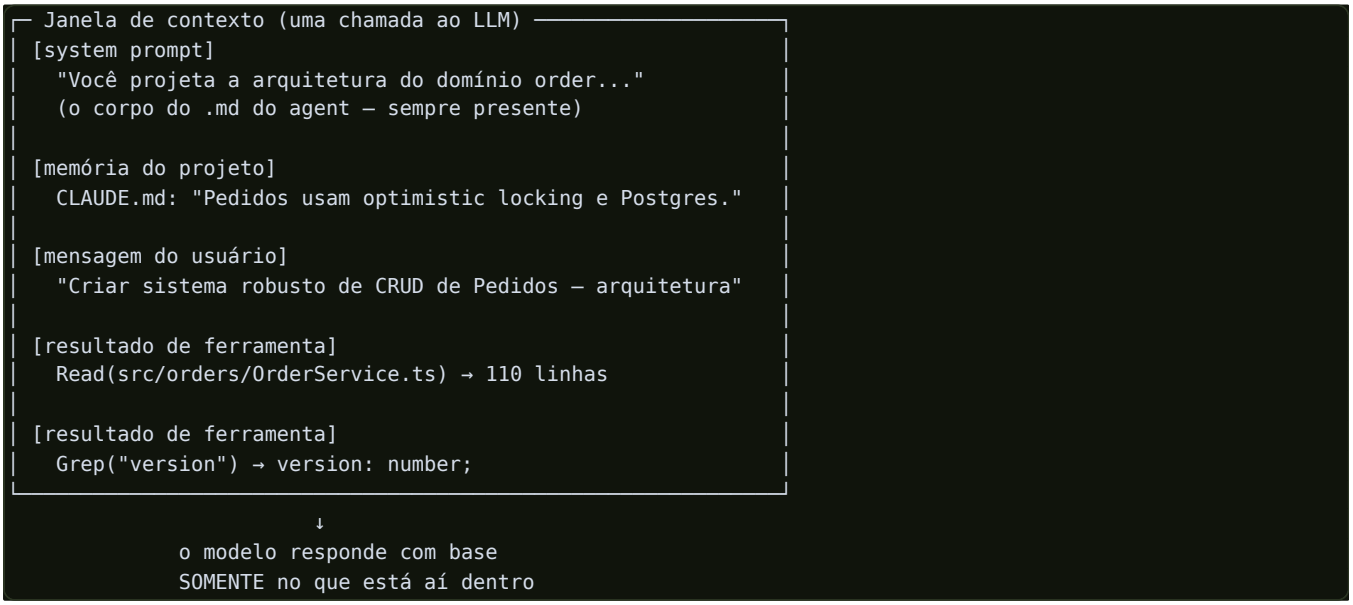
> Context é tudo que o modelo enxerga em uma única chamada. É um recurso finito. Engenheirar o que entra nele é metade do resultado.

TL;DR: Context é a memória de trabalho finita do modelo; curar o que entra — recuperar, isolar, compactar, persistir — é metade do resultado.

Nos capítulos anteriores, o `agent-order-architect` "leu o seu código" antes de opinar, e cada subagent rodou em "contexto isolado". Agora abrimos essa caixa: o que exatamente o modelo vê, por que isso é limitado, e como você controla.

Primeiro, o context em ação

Quando o `agent-order-architect` é acionado, o modelo não recebe "o seu projeto". Ele recebe uma montagem específica de texto — a janela de contexto daquela chamada:



Duas conclusões saltam:

1. **Se algo não está na janela, para o modelo não existe.** O `Grep("version")` encontrou a propriedade `version` — então o modelo conclui, corretamente, que há controle de versão ativo. A presença ou ausência são informações chave.
2. **A janela tem um tamanho máximo.** Não cabe o repositório inteiro. Alguém precisa escolher *quais* 110 linhas entram. Esse alguém é o harness, guiado pelo que o agent pede.

O que é o context

> O **context** (ou *janela de contexto*) é o conjunto total de tokens que o LLM recebe em uma chamada: o system prompt, a memória carregada, o histórico da conversa, os resultados de ferramentas e quaisquer arquivos recuperados. É a única coisa sobre a qual o modelo raciocina — e tem um limite rígido de tamanho.

Pense no context como a **memória de trabalho** (a RAM) do modelo, não o disco. O modelo não "lembra" do seu projeto entre chamadas; a cada chamada, o que importa precisa estar carregado na janela. Quando a chamada termina, some.

O que vive na janela

Parte	O que é	No nosso exemplo
System prompt	As instruções do agent (o corpo do `.md`)	"Você projeta a arquitetura…"
Memória	Fatos persistentes do projeto, carregados sempre	`CLAUDE.md`: optimistic locking, Postgres
Mensagens	O histórico da conversa atual	pedido do usuário + respostas
Resultados de ferramentas	Saída de `Read`, `Bash`, `Grep`, MCP…	conteúdo dos arquivos, saída de testes
Recuperação	Trechos buscados sob demanda	o `OrderService.ts`

A soma de tudo isso precisa caber no limite da janela. Estourou, algo é cortado ou compactado.

Context engineering: o trabalho de curar a janela

Como a janela é finita e o modelo raciocina só sobre ela, decidir **o que entra** é uma disciplina de engenharia. A Anthropic chama isso de *context engineering* — a evolução do "prompt engineering" quando o que importa não é só a frase que você escreve, mas todo o estado que o modelo recebe.

O princípio central: **sinal sobre ruído.** Tanto faltar informação quanto sobrar lixo degradam a resposta. Encher a janela com 50 arquivos "por garantia" não ajuda — piora, porque o modelo se distrai e perde o que importa no meio (o fenômeno *lost in the middle*, em que informação no meio de um contexto longo é menos aproveitada que no início ou no fim).

As técnicas que você já viu, agora nomeadas:

- **Recuperação seletiva.** Trazer só os arquivos relevantes (via `Grep` / `Glob` antes de `Read`), não o repositório inteiro. O architect achou `OrderService.ts` com um grep, não lendo tudo.
- **Isolamento por subagent (Capítulo 04).** Delegar mantém o contexto do orquestrador limpo: o backend mergulha em 6 arquivos no *seu* contexto; o pai só recebe o resumo.
- **Compactação.** Quando a conversa fica longa, o harness resume o histórico antigo para liberar espaço, preservando as decisões e descartando o ruído. É como o agente "anota o que importa e esquece o resto".
- **Memória persistente.** O que precisa sobreviver entre sessões não vive na conversa — vive em arquivos de memória. No Claude Code, o `CLAUDE.md` (ou `AGENTS.md`) carrega convenções e fatos do projeto em toda sessão, e há memória de longo prazo em arquivos dedicados. Foi assim que "locking, Postgres" apareceu na janela sem você redigitar.
- **Hooks que injetam contexto.** Eventos do harness podem inserir informação na janela automaticamente (por exemplo, lembrar uma regra antes de uma ferramenta rodar). Voltamos a hooks no Capítulo 07.

Como isso se conecta ao `agent`

A relação é dupla, e define a qualidade do agent:

> **O system prompt do agent é a parte permanente do contexto; tudo o mais é montado em volta dele a cada chamada.**

1. **O `.md` do agent ocupa contexto sempre.** Por isso um system prompt enxuto e preciso vale mais que um prolixo — cada token de instrução é um token a menos para o trabalho. O contrato em três seções (`o que entrega` / `como trabalha` / `restrições`) do Capítulo 03 é curto de propósito.
2. **As `tools` do agent moldam o que entra depois.** Dar `Grep, Glob` ao architect é o que permite a ele *encontrar* o arquivo certo antes de lê-lo — recuperação seletiva por design. Um agent sem ferramentas de busca tende a pedir arquivos errados e a entupir o contexto.
3. **`model` define o tamanho da janela disponível.** Modelos diferentes têm janelas diferentes; escolher o modelo (Capítulo 01) é também escolher quanto contexto cabe.

Em uma frase: **o agent é, em boa parte, uma decisão de context engineering congelada em um arquivo.** O que he instrui, o que ele pode buscar e qual cérebro ele usa determinam o que cabe na janela quando ele trabalha.

Trade-offs e armadilhas

- **Mais contexto não é melhor contexto.** Passar tudo “por segurança” dilui o sinal e custa mais tokens. Curar é melhor que despejar.
- **Contexto é dinheiro e latência.** Cada token na janela é cobrado e processado. Janelas enormes são lentas e caras — use o espaço com intenção.
- **O que não entra, não existe.** Bugs de agente frequentemente são bugs de contexto: a informação certa nunca chegou à janela. Antes de culpar o modelo, pergunte “ele viu o que precisava ver?”.
- **Compactação perde nuance.** Resumir libera espaço mas pode descartar um detalhe que importava. Decisões críticas merecem virar memória persistente, não ficar à mercê da compactação.
- **Lost in the middle.** Em contextos longos, ponha o mais importante no começo ou no fim, não enterrado no meio.

Como saber se você entendeu

Você dominou este capítulo se consegue:

- listar o que vive na janela de contexto;
- explicar o fenômeno *lost in the middle* e como mitigá-lo;
- justificar por que “o que não entra no contexto não existe para o modelo”.

Fontes

- Liu et al., “Lost in the Middle: How Language Models Use Long Contexts” (2023) — o estudo por trás do fenômeno citado: https://arxiv.org/abs/2307.03172 (https://arxiv.org/abs/2307.03172)
- Anthropic — “Effective context engineering for AI agents”: https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents (https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents)
- Anthropic — engenharia de prompt (boas práticas que valem para o contexto): https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/overview (https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/overview)
- Claude Code — memória (`CLAUDE.md`, memória de projeto e de usuário): https://code.claude.com/docs/en/memory (https://code.claude.com/docs/en/memory)

Síntese

Context é a memória de trabalho do modelo: finita, montada a cada chamada, e a única coisa sobre a qual ele raciocina. Engenheirar o que entra — recuperar o relevante, isolar por subagent, compactar o histórico, persistir o essencial em memória — é o que separa um agente que acerta de um que alucina por falta de informação. O agent, no fundo, é uma curadoria de contexto salva em disco.

Mas há um tipo de conhecimento que você não quer recarregar a cada chamada nem deixar morrer na compactação: o know-how reutilizável. Para isso existe uma camada própria.

Próximo: Capítulo 06 — A Skill.

Capítulo 06 — A Skill

Skills como conhecimento sob demanda com progressive disclosure e economia de tokens.

> Uma skill é conhecimento empacotado, carregado sob demanda. Ela estende um agent sem inchar o contexto dele — só entra na janela quando é relevante.

TL;DR: Skill é conhecimento empacotado, carregado sob demanda por *progressive disclosure* — estende o agent sem inchar o system prompt.

No capítulo anterior vimos que o contexto é finito e que o system prompt do agent ocupa espaço o tempo todo. Surge um problema: e o conhecimento especializado que o agent só precisa às vezes? Colocar tudo no prompt incha a janela em toda chamada. Não colocar nada deixa o agent ignorante. A skill resolve esse dilema.

Primeiro, a skill em ação

O `agent-order-architect` está desenhando o fluxo do CRUD de Pedidos (Orders) e chega no ponto crítico: concorrência e transições inválidas. Isso exige conhecimento específico — optimistic locking, controle de versão (`version`) e modelagem rígida de máquina de estados. Esse know-how **não está** no system prompt dele (que é curto). Ele está numa skill, que o agente carrega no momento certo:

```
[agent-order-architect]

A tarefa envolve concorrência e integridade de transição de status.
→ Carregando skill: improve-codebase-architecture

  (a skill traz para o contexto: padrões de concorrência com optimistic locking,
   regras estritas de transições de máquina de estados e logs de auditoria)

Com esse material, a decisão fica:
- Concorrência = optimistic locking via verificação incremental do campo `version`.
- Transições válidas = draft -> created -> pending -> paid | cancelled.
- Auditoria = gravar transição na tabela order_events no mesmo transaction block.
```

Antes dessa tarefa, o conhecimento de padrões de concorrência e integridade **não ocupava** nenhum espaço no contexto do agente. Ele entrou só quando a tarefa pediu, e poderia sair depois. Esse carregamento sob demanda é a essência da skill.

O que é uma skill

> Uma **skill** é uma capacidade empacotada — uma pasta com um arquivo `SKILL.md` (instruções + metadados) e, opcionalmente, recursos como scripts e documentos de referência — que o agente carrega **sob demanda**, quando a tarefa atual casa com a descrição da skill.

A anatomia mínima, no Claude Code:

```
.claude/skills/improve-codebase-architecture/
├── SKILL.md            # frontmatter (name, description) + instruções
├── reference.md        # material aprofundado (carregado só se preciso)
└── scripts/
    └── analyze.ts      # código que a skill pode rodar (ex.: TypeScript CLI)
```

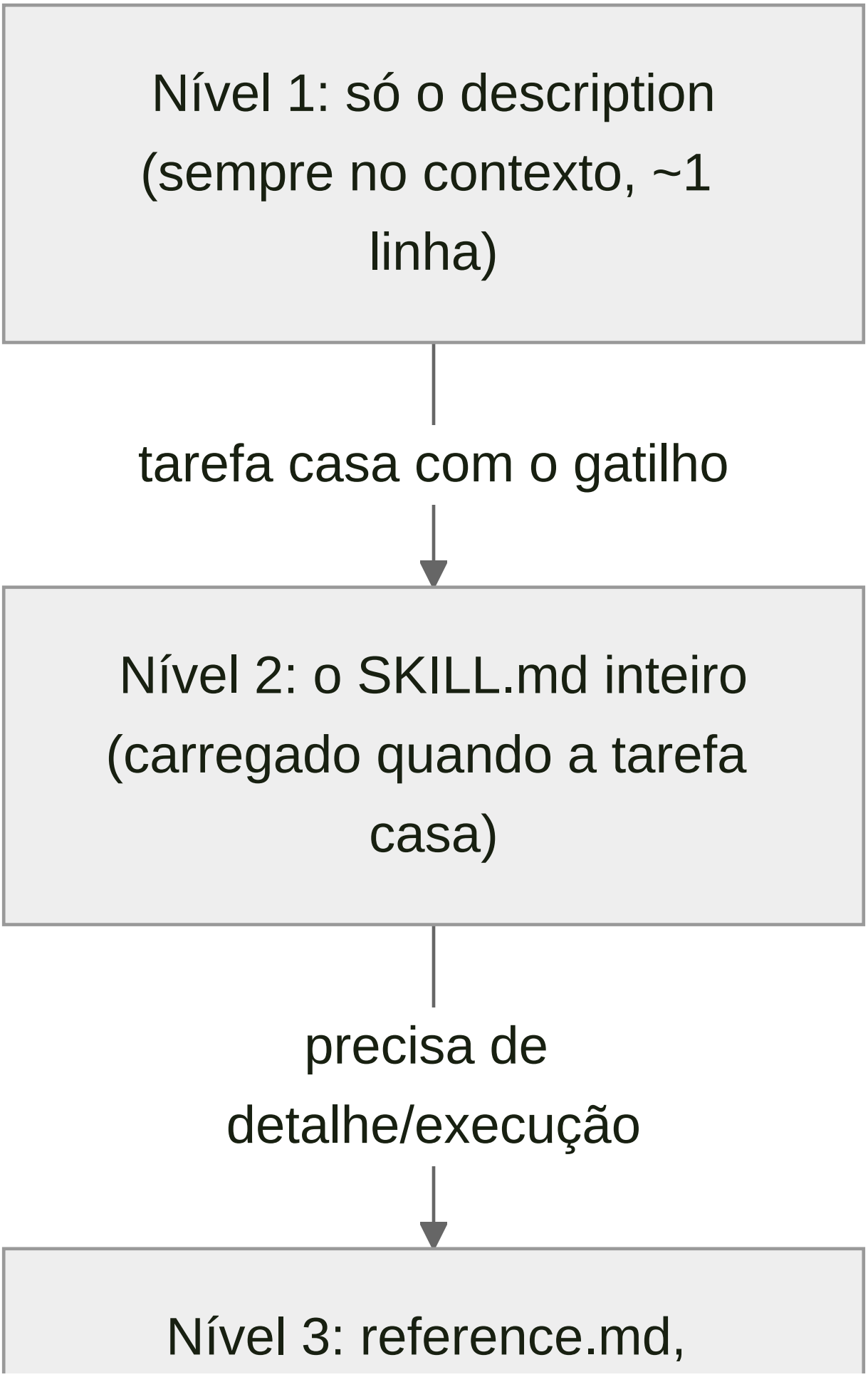
E o `SKILL.md` em si, no mesmo formato dos que já existem neste repositório:

```
---
name: improve-codebase-architecture
description: Encontra oportunidades de aprofundar a arquitetura de um
  código. Use ao melhorar arquitetura, achar refatorações, consolidar
  módulos acoplados ou tornar o código mais testável. Inclui padrões
  de concorrência, máquina de estados e auditoria de eventos.
---

# improve-codebase-architecture

## Padrões de Concorrência
- Controle concorrente com optimistic locking via versão (`version` check).

## Máquina de Estados
- Modele transições válidas de forma explícita; barre transições proibidas (ex. de cancelado para pago).

## Quando NÃO aplicar
- ...
```

Note: o formato é exatamente o que você já viu nas skills do repo — frontmatter com `name` e `description`, e um corpo em Markdown. A skill é, literalmente, conhecimento escrito como documento, com metadados que dizem quando usá-lo.

Progressive disclosure: o truque que torna skills baratas

Aqui está o mecanismo que faz skills funcionarem sem entupir o contexto — *progressive disclosure* (revelação progressiva), em três níveis:


Nível 1: só o description (sempre no contexto, ~1 linha)
tarefa casa com o gatilho
Nível 2: o SKILL.md inteiro (carregado quando a tarefa casa)
precisa de detalhe/execução
Nível 3: reference.md,

scripts
(carregados só se
realmente precisar)

- **Nível 1 — sempre presente, custo mínimo.** Só a `description` de cada skill fica no contexto o tempo todo. É como um índice: dezenas de skills custam pouquíssimos tokens, porque só os “títulos” estão carregados.
- **Nível 2 — sob demanda.** Quando a tarefa atual casa com a `description` de uma skill, o `SKILL.md` completo entra na janela. Foi o que aconteceu quando o architect chegou em “idempotência”.
- **Nível 3 — só se necessário.** Arquivos de referência e scripts dentro da pasta só são lidos se a tarefa exigir aquele detalhe. O agente carrega o `reference.md` apenas se o resumo do `SKILL.md` não bastar.

É o mesmo princípio do Capítulo 05 (sinal sobre ruído) aplicado ao conhecimento: você pode ter centenas de skills disponíveis, mas só paga o contexto das que de fato usa, no momento em que usa. Por isso a `description` da skill é tão crítica quanto a do agent — é ela que decide se a skill é “descoberta” na hora certa.

Skill, agent e subagent: quem é o quê

Esses três se confundem. A distinção:

Conceito	É um…	Tem estado/loop?	Reutilizável por…
Agent	trabalhador com um papel	sim (roda no harness)	—
Subagent	agent acionado por outro	sim	—
Skill	conhecimento/capacidade empacotada	não (é material, não trabalhador)	vários agents

A frase de bolso: **um agent é quem faz; uma skill é o que ele sabe.** Uma mesma skill (`tdd`, por exemplo) pode ser usada pelo `agent-order-backend` e pelo `agent-order-qa` — o conhecimento é compartilhado, os trabalhadores são distintos.

Skills vs Commands (Slash Commands)

Com a evolução das ferramentas CLI como o Claude Code, surge outra distinção vital: as **skills** vs. os **commands** (slash commands).

- **Skills:** São carregadas pelo harness de forma semântica e **implícita**. O agente lê o histórico de conversa, percebe a necessidade ("preciso de concorrência") e puxa o `SKILL.md` correspondente. É um fluxo transparente e orientado à cognição.
- **Commands (Slash Commands):** São chamados imperativos e **explícitos** disparados por você no terminal (ex.: `/goal`, `/schedule`, `/grill-me`). Eles iniciam fluxos de controle estruturados ou scripts determinísticos, sem depender do julgamento do modelo para ativação.

A frase de bolso: *Skills estendem a cognição do agente de forma implícita; Commands estendem a capacidade de operação da cabine (CLI) de forma explícita.*

Skills Auto-Melhoráveis: O Próximo Nível

Uma categoria emergente do ecossistema são as **Skills Auto-Melhoráveis** (self-improving skills). Um exemplo real do repositório é o harness de feedback-loop (ex.: `skill-recursiva-feedback-loop-harness`). Essas skills não contêm apenas instruções estáticas; elas contêm loops estruturados de auto-correção que instruem o agente a entrar em ciclos recursivos de:

1. Propor uma correção de código.
2. Rodar a suite de verificação/testes via tool (`Bash`).
3. Se falhar, consumir o log de erro, alimentar a própria memória de trabalho e ajustar recursivamente o código sem pedir intervenção humana.

Isso automatiza o fluxo de "Error Fix Loop" (EFL), minimizando a latência de desenvolvimento.

Recomendação CRÍTICA: Evite a Explosão de Tokens

> [!IMPORTANT] **O desenvolvedor deve testar o uso sem skills primeiro para evitar a explosão de tokens.**
>
> Embora skills sejam úteis, toda skill ativa consome o limite da janela de contexto (RAM) com tokens adicionais. Para tarefas triviais ou mecânicas, confie na inteligência bruta do modelo e nas diretrizes simples do `CLAUDE.md`. Ative ou crie skills apenas quando houver necessidade real de know-how denso e reutilizável.

Como isso se conecta ao `agent`

Lembra do frontmatter do Capítulo 03?

```
skills: [improve-codebase-architecture, diagnose]
```

Agora essa linha tem significado completo:

> **A skill é como você estende o que um agent sabe fazer sem inchar o system prompt dele.**

1. **O agent declara afinidade com skills.** O campo `skills` diz quais pacotes de conhecimento aquele papel costuma precisar. É uma dica de quais skills são relevantes para o trabalho dele.
2. **O conhecimento fica fora do prompt.** Em vez de despejar “como projetar concorrência” no system prompt do architect (custando contexto em toda chamada), isso vive na skill e entra só quando a tarefa pede. O prompt do agent continua curto; a competência, sob demanda.
3. **Skills são compartilhadas entre agents.** A squad inteira do Capítulo 04 referencia skills reais (`tdd`, `frontend-design`, `ui-ux-pro-max`...). Cada subagent puxa o conhecimento de que precisa, sem duplicar nada.

Skill está para conhecimento como subagent está para trabalho: as duas camadas existem para manter cada agent **focado e enxuto**, delegando o que não precisa estar sempre presente.

Trade-offs e armadilhas

- **A `description` é tudo.** Se ela não descreve bem *quando* usar a skill, a skill nunca é carregada — vira código morto. Escreva o gatilho com situações concretas, como no `description` do agent.
- **Skill não é agent.** Não tente colocar "comportamento autônomo" numa skill. Skill é material que um agent consome; quem age é o agent.
- **Excesso de skills polui o nível 1.** Cada skill custa sua `description` no contexto permanente. Centenas de skills mal descritas viram ruído de índice. Mantenha o catálogo curado.
- **Conhecimento desatualizado é pior que ausente.** Uma skill com um padrão obsoleto faz o agent errar com confiança. Skills precisam de manutenção como qualquer documentação.

Como saber se você entendeu

Você dominou este capítulo se consegue:

- explicar os três níveis do *progressive disclosure*;
- diferenciar skill, command, agent e subagent;
- justificar por que testar sem skills primeiro é uma prática recomendada de economia de tokens.

Fontes

- Anthropic — courses, exemplos oficiais de capacidades e padrões com Claude: https://github.com/anthropics/courses (https://github.com/anthropics/courses)
- Claude Code — Skills (estrutura, `SKILL.md`, progressive disclosure): https://code.claude.com/docs/en/skills (https://code.claude.com/docs/en/skills)
- Anthropic — "Equipping agents for the real world with Agent Skills": https://www.anthropic.com/news/agent-skills (https://www.anthropic.com/news/agent-skills)
- Claude Code — visão geral: https://code.claude.com/docs/pt/overview (https://code.claude.com/docs/pt/overview)

Síntese

Skill é conhecimento como documento, carregado sob demanda por *progressive disclosure*: só a descrição fica sempre no contexto; o conteúdo entra quando a tarefa casa. Isso permite que um agent enxuto tenha acesso a muita competência sem pagar o contexto dela o tempo todo. O `skills: [...]` do agent é a ponte entre o trabalhador e o que ele sabe.

Já temos agents, subagents, contexto e skills. Mas como você empacota tudo isso — uma squad inteira com suas skills, hooks e conexões — para instalar em outro repositório com um comando? Essa é a próxima camada.

Próximo: Capítulo 07 — O Plugin.

Capítulo 07 — O Plugin

Plugins como empacotamento distribuível de comandos, skills, hooks e MCP.

> Um plugin empacota agents, skills, comandos, hooks e configuração de MCP em uma unidade instalável. É como você distribui uma squad inteira entre times com um comando.

TL;DR: Plugin empacota agents, skills, comandos, hooks e MCP numa unidade versionada e instalável — é a unidade de *entrega* do que o agent precisa para funcionar em outro lugar.

Você construiu a squad `order`: nove agents, suas skills, suas convenções. Funciona no seu repositório. Agora o time de outro produto quer a mesma coisa. Como você entrega? Copiando dezenas de arquivos à mão? Há uma camada para isso.

Primeiro, o plugin em ação

Sem plugin, “compartilhar a squad” significa copiar `.claude/agents/*.md` (nove arquivos), `.claude/skills/*` (as skills referenciadas), os hooks, a configuração de MCP — e torcer para não esquecer nada nem versão. Com plugin, é um comando:

```
> /plugin marketplace add minhaempresa/order-squad
> /plugin install order-squad

Instalado order-squad v1.2.0:
  • 9 agents (orchestrator + 8 subagents)
  • 4 skills (concorrência, auditoria, ...)
  • 2 hooks (valida concorrência antes de commit; bloqueia segredo em log)
  • 1 servidor MCP (database inspector)
  • 1 comando: /order-new-field
```

Em uma linha, o outro time ganhou a squad completa, versionada, com tudo que ela precisa para funcionar. Atualizou? `/plugin update order-squad` e todos recebem a v1.3.0. Isso é distribuição de verdade, não copia-e-cola.

O que é um plugin

> Um **plugin** é um pacote distribuível que agrupa um ou mais componentes do Claude Code — agents, skills, comandos (*slash commands*), hooks e configuração de servidores MCP — sob um manifesto versionado, instalável a partir de um *marketplace*.

O plugin não é uma camada nova de capacidade. É uma camada de **empacotamento e distribuição** das camadas que você já conhece. Ele responde à pergunta “como eu levo isto para outro lugar de forma confiável?”.

Anatomia de um plugin

```
order-squad/
├── .claude-plugin/
│   └── plugin.json          # manifesto: nome, versão, descrição, autor
├── agents/                  # os 9 agents da squad
│   ├── agent-order-orchestrator.md
│   ├── agent-order-architect.md
│   └── ...
├── skills/                  # skills que a squad usa
│   └── improve-codebase-architecture/SKILL.md
├── commands/                # slash commands
│   └── order-new-field.md
├── hooks/                   # automações por evento
│   └── hooks.json
└── .mcp.json                # servidores MCP que a squad conecta (Cap. 08)
```

E o manifesto, `plugin.json`:

```
{
  "name": "order-squad",
  "version": "1.2.0",
  "description": "Squad completa do domínio order: orquestrador, 8 subagents, skills de concorrência e
máquina de estados.",
  "author": "minhaempresa"
}
```

O **marketplace** é só um repositório (git, por exemplo) com um catálogo desses plugins. `/plugin marketplace add <repo>` registra a fonte; `/plugin install <nome>` instala dali. É o mesmo modelo mental de um gerenciador de pacotes — `npm` / `cargo` para extensões do seu agente.

Hooks: a peça que o plugin frequentemente carrega

Mencionamos hooks de passagem nos capítulos 02 e 05. Aqui eles ganham forma, porque plugins são o jeito comum de distribuí-los.

> Um **hook** é um comando de shell que o harness dispara automaticamente em um evento do ciclo de vida — antes de uma ferramenta rodar (`PreToolUse`), depois (`PostToolUse`), quando a sessão tenta encerrar (`Stop`), entre outros.

Exemplo concreto, no `hooks.json` do plugin:

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          { "type": "command", "command": "scripts/bloqueia-segredo-em-log.sh" }
        ]
      }
    ]
  }
}
```

Esse hook roda antes de qualquer `Bash` e pode barrar um comando que vaze segredo em log. O ponto: hooks impõem política **deterministicamente**, fora do modelo. O LLM pode "querer" rodar algo perigoso; o hook decide se acontece — exatamente a fronteira de permissões do Capítulo 02, agora programável e distribuível dentro do plugin.

Como isso se conecta ao `agent`

A relação fecha a pergunta de distribuição:

> **O agent é o arquivo. O plugin é o formato de distribuição que leva esse arquivo — e tudo de que ele depende — para outro lugar, versionado.**

1. **O plugin empacota agents.** A squad `order` inteira (Capítulo 04) cabe num plugin. Quem instala recebe o orquestrador e os oito subagents prontos.
2. **E empacota o que os agents precisam.** Um agent sozinho não basta: ele referencia skills (Capítulo 06), conecta servidores MCP (Capítulo 08) e depende de hooks de política. O plugin junta tudo isso, então o agent funciona igual no destino.
3. **Versiona o conjunto.** `agent-order-architect` v1.2.0 vem com as skills e a configuração de MCP daquela versão. Atualizar o plugin atualiza o conjunto coerente, não peças soltas que podem divergir.

Em uma frase: **se o agent é a unidade de trabalho, o plugin é a unidade de entrega.**

Trade-offs e armadilhas

- **Instalar plugin é confiar.** Um plugin pode trazer hooks (comandos de shell) e servidores MCP (acesso a sistemas). Instalar um plugin de terceiro é rodar o código dele na sua máquina. Trate marketplaces como você trata dependências: audite a fonte.
- **Versão acopla componentes.** A vantagem (conjunto coerente) é também o risco: um bug numa skill do plugin afeta todos os agents que a usam. Versione com cuidado e teste antes de subir.
- **Marketplace exige curadoria.** Um catálogo cheio de plugins redundantes ou mal descritos vira ruído. Mantenha o seu enxuto e bem documentado.
- **Nem tudo precisa ser plugin.** Para um agent usado só no seu repo, o arquivo em `.claude/agents/` basta. Plugin é para o que você *distribui*.

Como saber se você entendeu

Você dominou este capítulo se consegue:

- listar o que um plugin pode empacotar;
- explicar o papel de um hook `PreToolUse` na fronteira de permissões;
- justificar por que instalar um plugin de terceiros é uma decisão de confiança.

Fontes

- Anthropic — anthropic-cookbook, exemplos de extensão e automação com Claude: https://github.com/anthropics/anthropics-cookbook (https://github.com/anthropics/anthropics-cookbook)
- Claude Code — Plugins (estrutura, `plugin.json`, instalação): https://code.claude.com/docs/en/plugins (https://code.claude.com/docs/en/plugins)
- Claude Code — Plugin marketplaces (publicar e instalar): https://code.claude.com/docs/en/plugin-marketplaces (https://code.claude.com/docs/en/plugin-marketplaces)
- Claude Code — Hooks (eventos, `PreToolUse` / `PostToolUse` / `Stop`): https://code.claude.com/docs/en/hooks (https://code.claude.com/docs/en/hooks)

Síntese

Plugin é a camada de empacotamento: ele pega agents, skills, comandos, hooks e configuração de MCP e os transforma em uma unidade versionada e instalável, distribuída por um marketplace. É o que faz a sua squad `order` deixar de ser “um monte de arquivos no meu repo” e virar “um pacote que qualquer time instala em um comando”.

Sobrou uma dependência que o plugin empacota mas ainda não explicamos: como o agent fala com o mundo externo — o Postgres, o banco de dados de produção de verdade. Esse é o protocolo da próxima camada.

Próximo: Capítulo 08 — O MCP.

Capítulo 08 — O MCP

MCP como protocolo para conectar agentes a dados e ações externas.

> MCP (Model Context Protocol) é o padrão aberto que conecta o agente ao mundo externo — o banco de dados Postgres, APIs externas e sistemas legados — por uma interface única, em vez de uma integração ad hoc para cada sistema.

TL;DR: MCP é o padrão aberto que dá ao agente uma porta única para sistemas externos — um servidor embrulha o Postgres ou APIs externas e expõe ferramentas ao loop do agente.

Até aqui, as ferramentas do agente agiam na sua máquina: `Read`, `Edit`, `Bash`, `Grep`. Mas o `agent-order-backend` precisa saber se as tabelas do banco de dados *de verdade* estão com o schema correto em produção, e o `agent-order-analytics` precisa consultar dados consolidados de pedidos. Isso é fora da máquina local. Como o agente alcança esses sistemas? Com MCP.

Primeiro, o MCP em ação

O `agent-order-architect` quer validar uma suposição: o schema da tabela de pedidos no banco de dados local ou de staging já suporta optimistic locking? Em vez de você abrir um cliente SQL e rodar queries, o agente consulta direto, por uma ferramenta MCP:

```
[agent-order-architect]

Verificando o schema da tabela de pedidos no banco de dados…
→ mcp__postgres__describe_table(table_name: "orders")
  ← columns: [
      { name: "id", type: "uuid" },
      { name: "status", type: "varchar" },
      { name: "version", type: "integer" }
    ]

Confirmado: a tabela possui a coluna "version" necessária para a implementação do optimistic locking no
service do backend.
```

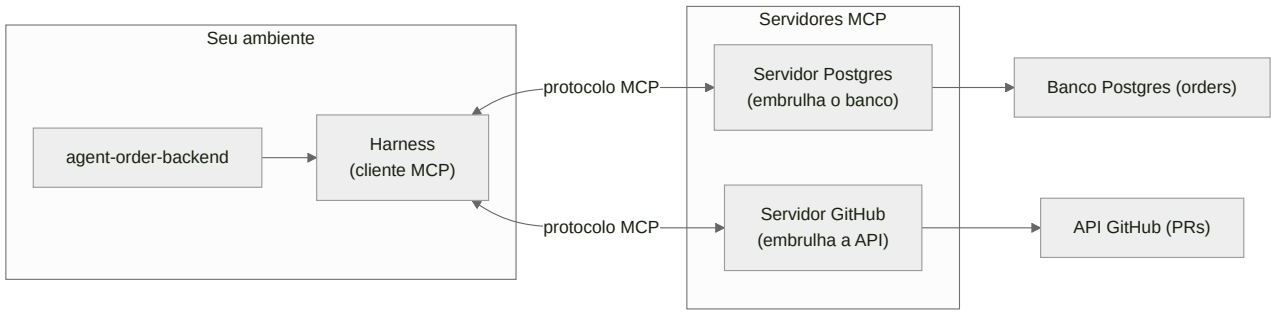
A chamada `mcp__postgres__describe_table()` não é um arquivo local nem um comando de shell direto. É uma ferramenta exposta por um **servidor MCP** que embrulha a conexão ao banco Postgres. Para o agente, ela aparece como mais uma ferramenta no loop do Capítulo 02 — ele pede, o harness executa, o resultado volta para a janela de contexto. A diferença é que, por baixo, isso falou com o banco de dados por um protocolo padronizado.

O que é o MCP

> O **MCP (Model Context Protocol)** é um padrão aberto, publicado pela Anthropic em novembro de 2024, para conectar aplicações de LLM a fontes de dados e ferramentas externas. Ele define como um *cliente* (o harness) conversa com *servidores* que expõem ferramentas, dados e prompts de forma uniforme.

A analogia que a própria Anthropic usa: **MCP é como uma porta USB-C para aplicações de IA.** Antes do USB-C, cada aparelho tinha seu conector; integrar N ferramentas a M aplicações dava N×M integrações sob medida. Com um padrão, cada lado implementa a interface uma vez e tudo conversa. MCP faz isso para conectar agentes a sistemas: o Postgres expõe um servidor MCP uma vez, e qualquer harness compatível passa a usá-lo — sem código de integração específico.

Como funciona: cliente e servidor



- **Cliente MCP**: o harness (Claude Code). Ele descobre quais ferramentas cada servidor oferece e as apresenta ao modelo.

- **Servidor MCP**: um processo que expõe capacidades por três tipos de primitiva:
  - **Tools** — ações que o modelo pode chamar (ex.: `run_query`, `describe_table`). É o que mais usamos.
  - **Resources** — dados que o servidor disponibiliza para leitura (ex.: logs ou schemas de banco).
  - **Prompts** — modelos de prompt reutilizáveis que o servidor sugere.
- **Transporte**: cliente e servidor falam por `stdio` (processo local) ou HTTP (servidor remoto). O protocolo é o mesmo; muda só o canal.

Como você conecta um servidor

A configuração vive num arquivo (`.mcp.json`), e o plugin do Capítulo 07 pode trazê-la pronta:

```
{
  "mcpServers": {
    "postgres": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-postgres"],
      "env": { "PG_CONNECTION_STRING": "${PG_CONNECTION_STRING}" }
    }
  }
}
```

Conectado o servidor, suas ferramentas aparecem para o agente com o prefixo `mcp__<servidor>__<ferramenta>` — foi de onde veio o `mcp__postgres__describe_table` do exemplo. Do ponto de vista do agente, são apenas mais ferramentas no loop.

Como isso se conecta ao `agent`

A relação amarra de volta ao Capítulo 02 (ferramentas) e ao Capítulo 03 (o campo `tools`):

> **MCP estende o conjunto de ferramentas do agent para além da máquina local. O que `Read` e `Bash` são para arquivos e shell, as ferramentas MCP são para sistemas externos.**

1. **As ferramentas MCP entram no loop como qualquer outra.** O agente não sabe (nem precisa saber) que `mcp__postgres__run_query` fala com um servidor remoto. Para ele, é uma ferramenta — pensar → chamar → observar, igual ao Capítulo 02.
2. **O `tools` do agent controla o acesso.** Você pode dar ao `agent-order-analytics` acesso ao servidor MCP do Postgres (leitura), mas negar a ele ferramentas de alteração. O princípio do menor privilégio do Capítulo 03 vale também — e principalmente — para ferramentas que tocam sistemas reais.
3. **O plugin distribui a conexão.** A squad `order` empacotada (Capítulo 07) já vem com o `.mcp.json` do database inspector. Quem instala o plugin ganha o agente *e* a porta para o banco de dados que ele precisa.

MCP vs CLI: Raciocinando sobre Performance e Economia de Tokens

Uma dúvida frequente é se devemos expor uma capacidade local como um **servidor MCP** ou como uma ferramenta **CLI nativa** executada via shell (`Bash`).

Aqui reside uma regra de ouro de performance e token economy: **Use o CLI local para operações locais de desenvolvimento (compilar, rodar testes, lint).**

- **Explosão de contexto e latência:** Embrulhar um compilador (como o `tsc`) em um servidor MCP adiciona camadas de serialização JSON, latência de IPC local, e inunda a janela de contexto com o overhead do protocolo de metadados do MCP.
- **Superioridade do CLI:** Deixar o agente rodar os testes e linters diretamente via terminal/shell (`tools: Bash`) aproveita o processo do OS nativo. É infinitamente mais rápido, consome menos recursos, e preserva a janela de contexto mantendo apenas a saída essencial.
- **Quando usar o MCP:** O MCP brilha na conexão de sistemas **externos** (como APIs de terceiros, painéis SaaS, bancos de dados corporativos) que não possuem uma interface CLI local acessível de forma trivial ou segura.

A frase de bolso: *O CLI é para as ferramentas do próprio desenvolvedor no ambiente de trabalho local; o MCP é a porta USB-C para o ecossistema externo.*

Em uma frase: **MCP é como o agent deixa de raciocinar só sobre o seu código e passa a agir sobre o seu negócio.**

Trade-offs e armadilhas

- **Conectar um servidor MCP é conceder acesso.** Um servidor com credenciais de escrita no banco de produção pode apagar dados. Trate credenciais de MCP como você trata qualquer segredo de produção: menor privilégio, escopo restrito, rotação.
- **Injeção de prompt via resultado de ferramenta.** O texto que um servidor MCP devolve entra na janela de contexto e o modelo pode segui-lo como instrução. Um servidor comprometido (ou um dado malicioso no banco) pode tentar "instruir" o agente. Não confie cegamente no que volta de uma ferramenta externa.
- **Latência e disponibilidade.** Cada chamada MCP é uma ida à rede ou a outro processo. Servidor lento ou fora do ar trava o loop do agente. Considere timeouts e fallback.
- **Confiança no servidor.** Use servidores MCP oficiais ou auditados. Um servidor de terceiro roda no seu ambiente e vê o que você passa a ele.

Como saber se você entendeu

Você dominou este capítulo se consegue:

- explicar a relação cliente (harness) ↔ servidor MCP;
- justificar a diferença entre usar uma ferramenta CLI local vs. um servidor MCP;
- citar dois riscos de segurança de conectar um servidor MCP.

Fontes

- Model Context Protocol — servidores de referência (implementações oficiais open source): https://github.com/modelcontextprotocol/servers (https://github.com/modelcontextprotocol/servers)
- Anthropic — anúncio do MCP (novembro de 2024): https://www.anthropic.com/news/model-context-protocol (https://www.anthropic.com/news/model-context-protocol)
- Especificação e documentação do MCP (padrão aberto): https://modelcontextprotocol.io (https://modelcontextprotocol.io)
- Claude Code — MCP (configurar servidores, `.mcp.json`): https://code.claude.com/docs/en/mcp (https://code.claude.com/docs/en/mcp)

Síntese

MCP é o padrão aberto que dá ao agente uma porta única para o mundo externo: em vez de uma integração sob medida para cada sistema, um servidor MCP embrulha o Postgres ou APIs externas e expõe ferramentas que entram no loop do agente como qualquer outra. É o que transforma um agente que conhece o seu código num agente que opera o seu banco de dados — com todo o cuidado de segurança que essa ponte exige.

Temos agora todas as camadas conceituais. Falta o lugar concreto onde você opera tudo isso, no dia a dia, no terminal.

Próximo: Capítulo 09 — O CLI.

Capítulo 09 — O CLI

CLI como cabine de comando para workflows agênticos no host local.

> O CLI é onde você opera todas as camadas anteriores: o terminal como cabine de comando do harness. É onde o agente encontra o seu ambiente real — arquivos, git, shell, testes.

TL;DR: O CLI é a cabine onde você opera todas as camadas — aciona agents, troca de modelo, instala plugins, conecta MCP — e, em modo headless, vira etapa de pipeline.

Falamos de LLM, harness, agent, subagent, context, skill, plugin e MCP como conceitos. Mas em que lugar concreto você *roda* tudo isso? Para a maioria dos desenvolvedores, a resposta é a linha de comando. O Claude Code é, na sua forma pri## Primeiro, o CLI em ação

Você abre o terminal no repositório do e-commerce e digita:

```
$ claude
```

A sessão começa, e tudo dos capítulos anteriores acontece aqui:

```
> Criar sistema robusto de CRUD de Pedidos. Quero a arquitetura antes do código.

• Acionando agent-order-architect (model: opus)        ← agent + LLM
• Read(src/orders/OrderService.ts)                       ← harness + tools
• mcp__postgres__describe_table() → [orders, events]     ← MCP
• Carregando skill improve-codebase-architecture         ← skill
• [hook PreToolUse: validar-concorrencia.sh] ok          ← plugin/hook

Decisões de arquitetura: [...]
```

Cada linha dessa sessão é uma camada deste e-book em operação — e o CLI é o lugar onde elas se encontram. O terminal não é um detalhe de interface: é onde o agente toca o seu ambiente de desenvolvimento de verdade.

O que é o CLI

> Um **CLI** (*Command-Line Interface*, interface de linha de comando) é um programa operado por texto no terminal. No nosso contexto, o Claude Code é um CLI: o ponto de entrada pelo qual você inicia uma sessão, conversa com o agente e o deixa agir sobre o seu projeto.

O Claude Code também existe em outras formas — extensão de IDE (VS Code, JetBrains), app de desktop e web. Mas o CLI é a forma canônica, e por um bom motivo: é onde o desenvolvimento de verdade já acontece. O agente que vive no terminal está a um `Bash` de distância dos seus testes, do seu `git`, do seu build.

Por que o terminal importa

O CLI dá ao harness acesso direto ao ambiente onde você trabalha:

- **Sistema de arquivos**: ler e editar o código no lugar, sem upload.
- **Shell**: rodar testes, build, linters, `git` — as ferramentas `Bash` do Capítulo 02 operam no seu shell real.
- **Composição Unix**: a saída do agente pode ser canalizada (`|`) para outras ferramentas, e ele pode consumir a saída delas.
- **Versionamento**: como agents, skills e plugins são arquivos, eles entram no `git` junto com o código — revisados em pull request como qualquer mudança.

Comandos Customizados (Slash Commands)

O CLI permite a você interagir de forma imediata via **Slash Commands** (como `/goal`, `/schedule`, `/grill-me`). Mais do que usar os comandos padrão, você pode **customizar e estender** a cabine de comando criando novos atalhos no repositório. Esses comandos são definidos salvando arquivos Markdown em `.claude/commands/<nome-do-comando>.md`. Quando digitados, eles forçam uma instrução específica ou script determinístico diretamente na sessão, pulando o loop cognitivo do assistente geral.

Git Worktrees: Isolamento Total para Operações Agênticas

Uma das melhores práticas avançadas ao operar agentes no terminal é o uso de **Git Worktrees**. Muitas vezes, uma tarefa agêntica longa (como um `/goal` para reescrever um CRUD inteiro) pode durar minutos e travar o seu repositório local, impedindo você de trabalhar em outros arquivos ou ramos paralelos.

- **Como funciona:** O Git Worktree permite que você crie um diretório físico totalmente separado no disco, mas conectado ao mesmo repositório git (`git worktree add ../orders-refactor feat/orders-refactor`).
- **Vantagem para Agentes:** Você pode inicializar a sessão do CLI do agente apontando para esse worktree isolado. O agente pode fazer alterações profundas, rodar scripts agressivos de compilação, quebrar códigos intermediários e testar em background, enquanto o seu diretório de trabalho principal (`main` ou `develop`) continua limpo e disponível para seu fluxo de trabalho pessoal.

O CLI como cabine das camadas anteriores

O que você aprendeu nos capítulos anteriores aparece, no CLI, como controles concretos:

Camada	Como aparece no CLI
LLM (Cap. 01)	`/model` troca o cérebro da sessão (opus/sonnet/haiku) — model routing na prática.
Harness (Cap. 02)	É o próprio Claude Code rodando o loop; modos de permissão controlam o que ele executa.
Agent (Cap. 03)	Acionado automaticamente pelo `description`, ou invocado quando você descreve a tarefa.
Subagent (Cap. 04)	O orquestrador delega dentro da mesma sessão; você vê cada subagent reportar.
Context (Cap. 05)	`CLAUDE.md` é carregado a cada sessão; a compactação acontece quando a conversa cresce.
Skill (Cap. 06)	Carregada sob demanda; comandos `/` vêm de skills e plugins.
Plugin (Cap. 07)	`/plugin install` adiciona agents, skills, hooks e comandos à sua sessão.
MCP (Cap. 08)	Servidores do `.mcp.json` sobem com a sessão; suas ferramentas viram `mcp__…`.

O CLI é o denominador comum: o lugar onde todas essas peças, definidas em arquivos, ganham vida em uma conversa.

Modo headless: o CLI sem você na frente

Há um recurso do CLI que destrava automação séria — o **modo headless** (não interativo). Em vez de uma conversa, você passa o pedido como argumento e captura a saída:

```
$ claude -p "Rode os testes de orders e resuma as falhas" > relatorio.txt
```

Isso transforma o agente em um passo de pipeline. Você pode chamá-lo de um script, de um hook de `git`, de um job de CI — o mesmo agente que você usa interativamente, agora dentro de uma automação. É a ponte entre “assistente que eu converso” e “etapa programável do meu fluxo”.

Como isso se conecta ao `agent`

O fechamento do arco:

> **O CLI é a cabine onde você invoca e opera o agent. Se o agent é o especialista e o harness é o motor, o CLI é o painel de controle.**

1. **É onde o agent é acionado.** Você descreve a tarefa no terminal; o harness casa com o `description` e aciona o `agent-order-architect`. O ciclo inteiro dos capítulos anteriores começa com você digitando no CLI.
2. **É onde você ajusta a configuração ao vivo.** `/model opus` antes de uma decisão difícil; `/plugin install order-squad` para trazer a squad; conectar um servidor MCP. O CLI é onde as decisões dos capítulos 01–08 viram comandos.
3. **É onde o agent vira automação.** Em modo headless, o mesmo `agent-order-architect` pode rodar num CI a cada PR, revisando a arquitetura automaticamente. O agente sai do terminal interativo e entra no pipeline.

Em uma frase: **o CLI é onde o e-book inteiro deixa de ser teoria e vira o seu fluxo de trabalho.**

Trade-offs e armadilhas

- **Poder no terminal exige permissões sérias.** Um agente com `Bash` no seu shell pode fazer estrago. Use os modos de permissão; não rode tudo em "aceitar tudo" por preguiça.
- **Headless remove o humano do loop.** Automação é ótima até o agente fazer a coisa errada sem ninguém olhando. Em CI, restrinja ferramentas e dê escopo apertado — o menor privilégio do Capítulo 03 vale dobrado sem supervisão.
- **CLI tem curva.** Para quem vive em GUI, o terminal assusta no começo. Mas é justamente a integração com shell/git/arquivos que dá ao agente o acesso que o torna útil.
- **Saída para pipeline precisa ser estável.** Em scripts, prefira formatos previsíveis e trate o agente como um comando que pode falhar — com timeout e verificação do resultado.

Como saber se você entendeu

Você dominou este capítulo se consegue:

- mapear pelo menos quatro camadas anteriores aos controles do CLI;

- explicar o que o modo headless habilita e quais riscos ele traz;
- descrever como os Git Worktrees auxiliam no isolamento de sessões agênticas longas.

Fontes

- Anthropic — anthropic-sdk-typescript, o SDK oficial para automações e integrações TypeScript/JS: https://github.com/anthropics/anthropic-sdk-typescript (https://github.com/anthropics/anthropic-sdk-typescript)
- Claude Code — referência do CLI (comandos, flags, modo headless): https://code.claude.com/docs/en/cli-reference (https://code.claude.com/docs/en/cli-reference)
- Claude Code — visão geral (formas: CLI, IDE, desktop, web): https://code.claude.com/docs/pt/overview (https://code.claude.com/docs/pt/overview)
- Claude Code — Git Worktrees: https://code.claude.com/docs/pt/worktrees (https://code.claude.com/docs/pt/worktrees)

Síntese

O CLI é onde tudo se encontra: o terminal como cabine de comando do harness, com acesso direto aos arquivos, ao shell, ao git e aos testes do seu projeto. É lá que você aciona agents, troca de modelo, instala plugins, conecta MCP — e, em modo headless, transforma o agente em uma etapa do seu pipeline. As camadas anteriores são definições em arquivos; o CLI é onde elas viram trabalho feito.

Temos o stack inteiro. No último capítulo, montamos tudo de uma vez e seguimos a tarefa de CRUD de Pedidos da primeira à última camada, sem cortes.

Próximo: Capítulo 10 — Síntese.

Capítulo 10 — Síntese

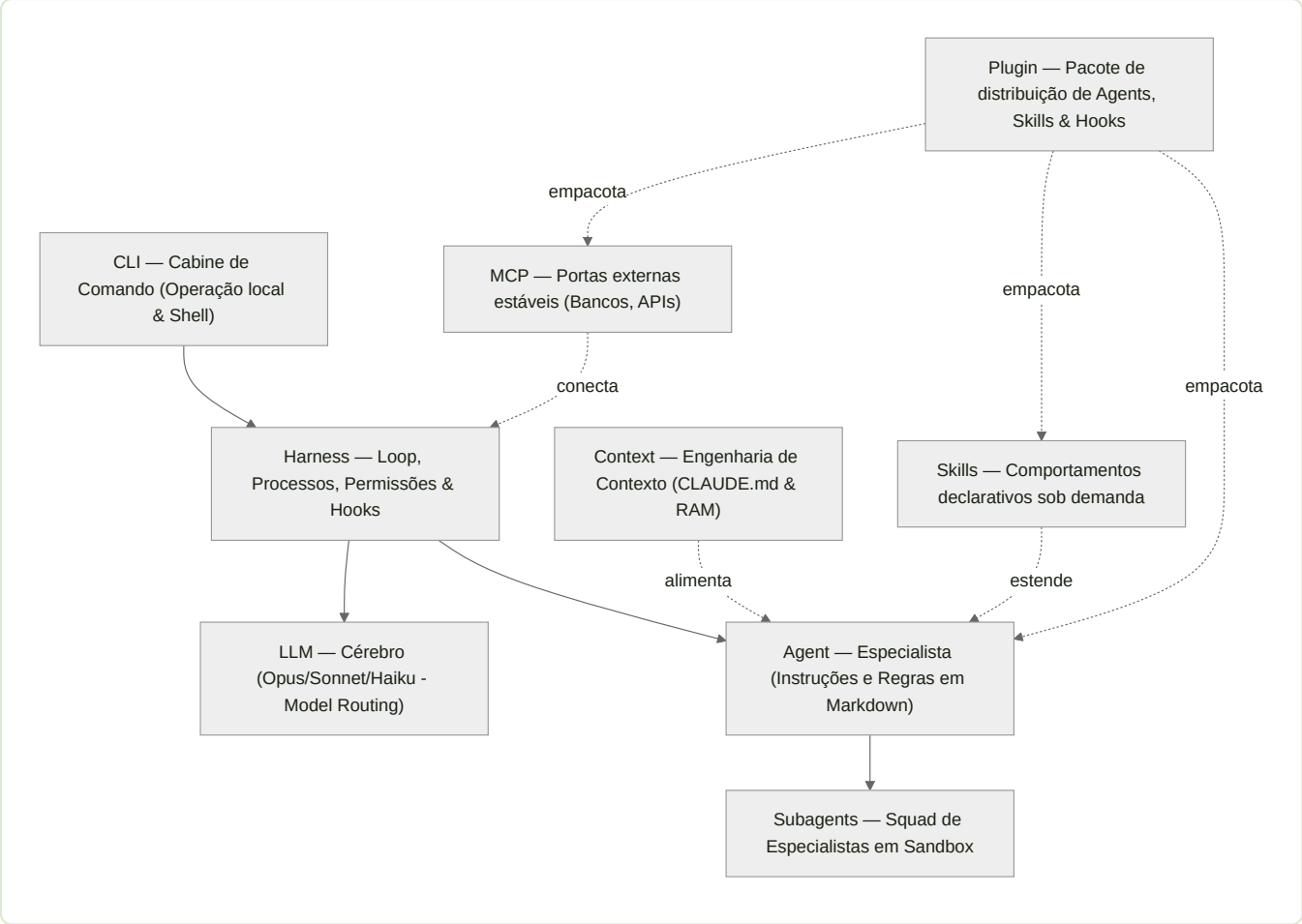
Síntese do stack AI-native completo em um modelo mental integrado.

> Nove camadas, um ecossistema. Aqui consolidamos todo o stack sob uma tarefa de CRUD de Pedidos real, rastreando o fluxo do primeiro token ao deploy, e mapeando as ferramentas que definem o estado da arte do desenvolvimento agêntico.

TL;DR: O stack completo opera de forma coesa para entregar um CRUD de Pedidos com isolamento via Git Worktrees, otimização de tokens com proxies locais (como RTK/Caveman), automações com Slash Commands e Skills recursivas, e feedbacks de ambiente em tempo real com Hooks e notificações do OS.

Você começou este e-book com uma lista desordenada de termos: LLM, harness, agent, subagent, context, skill, plugin, MCP, CLI. O objetivo deste trabalho foi transformar esse vocabulário solto em um sistema arquitetural estável: peças interdependentes onde cada camada resolve exatamente o gargalo que a camada anterior não consegue cobrir.

O stack em uma visão sistêmica



Analisando a topologia do fluxo de execução:

- **Fluxo Descendente (Comando)**: Você digita no **CLI**, que aciona o **harness** local. O harness inicializa a sessão carregando o **context** otimizado, interpretando os **plugins** instalados, estendendo capacidades através de **skills** e abrindo conexões **MCP** declaradas. O harness então invoca o **agent** correto, que gerencia e delega tarefas complexas a **subagents** especializados, orquestrando as chamadas de inferência no **LLM** adequado.
- **Fluxo Ascendente (Execução)**: O **LLM** gera a decisão lógica que o **harness** intercepta, valida com **hooks** determinísticos locais e executa no ambiente nativo do seu repositório via **CLI**, garantindo segurança e controle.

O CRUD de Pedidos pelo fluxo agêntico

Vamos traçar a jornada de ponta a ponta para implementar um **CRUD de Pedidos (Orders)** robusto — completo com tratamento de concorrência por optimistic locking e histórico de transições de estados — mapeando exatamente onde e por que cada camada entra em ação.

1. Preparando a cabine com Git Worktrees no CLI

Antes de iniciar a sessão com o agente, você quer evitar que arquivos temporários, builds inacabados ou testes quebrados interrompam seu trabalho paralelo no ramo principal. Você inicializa um **Git Worktree** dedicado para a tarefa:

```
$ git worktree add ../orders-sandbox feat/orders-crud
$ cd ../orders-sandbox
$ claude
```

O **CLI** do Claude Code é inicializado nesse diretório limpo e isolado. O CLI apresenta uma vantagem técnica gritante em relação a interfaces web ou extensões genéricas de IDE: **o CLI atua diretamente no host, acessando processos de shell nativos**. Isso significa que ferramentas de linters (`eslint`), testes unitários (`jest`/`vitest`), compiladores (`tsc`) e comandos de versionamento (`git`) rodam sem o overhead de serialização, rede ou latência inerentes aos protocolos de rede. É a superioridade de performance e tokens do CLI em ação: chamadas diretas de shell local poupam milhares de tokens que seriam desperdiçados descrevendo o estado do sistema de arquivos pela API.

```
> /goal Implementar o CRUD de Pedidos com controle de concorrência e fila de eventos.
```

2. O Harness e a Curadoria de Contexto

Ao ler a instrução, o **harness** é ativado. Ele carrega as diretrizes do arquivo `CLAUDE.md` e do indexador de instruções `AGENTS.md` do projeto.

A engenharia de contexto atua aqui como uma memória RAM limpa. Em vez de enviar toda a árvore de arquivos ao LLM, o harness usa indexação seletiva. O sinal-sobre-ruído é mantido alto: apenas a convenção de nomenclatura de TypeScript, o padrão de design do repositório e os schemas SQL relevantes das tabelas `orders` e `order_events` são puxados para a janela ativa.

3. O Agent Especialista e a Squad de Subagents

O harness avalia as descrições dos agentes disponíveis e aciona o especialista `agent-order-architect`. Roteado automaticamente para rodar no Claude 3.5 Sonnet ou no Claude 3 Opus (devido à complexidade do desenho de concorrência), o arquiteto analisa a tarefa e percebe que codificar, testar e documentar tudo de uma vez estourará o limite cognitivo de uma única sessão.

Ele decide criar uma **squad de subagents** isolados em sandboxes dedicadas:

1. **`product-manager`**: Analisa as regras de negócio para as transições de estado (`draft` -> `created` -> `pending_payment` -> `paid` -> `cancelled`) e gera os critérios de aceitação.
2. **`backend-coder`**: Escreve o serviço em TypeScript (`OrderService.ts`), aplicando concorrência otimista com verificação de versão da linha (`version` + 1) e persistência das transições na tabela `order_events`.
3. **`qa-tester`**: Gera testes automatizados de integração concorrente simulando requisições paralelas para garantir que nenhuma transição de estado seja sobrescrita indevidamente.

Cada subagent trabalha com ferramentas limitadas (princípio do menor privilégio) e relata apenas o resultado compilado ao orquestrador, mantendo o contexto principal limpo.

4. Custom Slash Commands vs. Skills no Loop

Durante a implementação, o desenvolvedor pode interagir com a sessão usando atalhos rápidos. É crucial entender a diferença de finalidade de cada um:

- **Slash Commands (Comandos Customizados)**: São atalhos de automação focados no fluxo do usuário (salvos em `.claude/commands/<nome>.md`, como `/goal` ou `/grill-me`). Eles forçam um comportamento determinístico de controle de sessão ou injetam prompts estruturados de forma imediata na pilha de execução.
- **Skills (Conhecimento Comportamental)**: São arquivos de habilidades declarativas salvas em `.claude/skills/<nome>/SKILL.md` que o agente carrega de forma dinâmica somente se identificar que a tarefa atual necessita daquela habilidade.

Para garantir a qualidade, o agente carrega a skill recursiva de auto-melhoria `skill-recursiva-feedback-loop-harness`. Esta skill executa um ciclo autônomo (EFL - Error Fix Loop): compila o código TypeScript com `npx tsc --noEmit`, roda os testes com `npm run test` e, se encontrar falhas, lê as mensagens de erro, ajusta o código e repete o ciclo recursivamente até que a execução esteja livre de erros e em total conformidade técnica.

> [!WARNING] **Aviso Crítico de Economia**: Sempre teste a execução básica do seu código *sem skills* ativas primeiro. Carregar múltiplas skills pesadas de forma contínua em tarefas simples gera uma explosão de tokens desnecessária, consumindo rapidamente a cota de processamento. Use skills complexas apenas quando a automação de correção de falhas for estritamente necessária.

5. Gateways Externos via MCP

Para persistir os dados e disparar notificações reais para sistemas de terceiros, o harness se conecta a servidores locais e remotos usando o **Model Context Protocol (MCP)**.

- O agente utiliza a ferramenta do MCP de banco de dados (`mcp__postgres__query`) para inspecionar os índices das tabelas e garantir que o campo `order_id` na tabela `order_events` possui a indexação correta para buscas de auditoria rápidas.
- O agente utiliza o MCP de mensageria para validar a fila de transição de status no sistema de mensagens externas.

O MCP atua como uma interface padronizada estável para o agente acessar o mundo externo sem precisar codificar conexões brutas em nível de soquete em cada execução.

6. Hooks e Notificações de Ambiente

Antes de consolidar as alterações, o harness aciona os **Hooks** de pré-execução declarados no manifesto do plugin do projeto:

- O hook `PreToolUse` executa um script bash local (`validar-typescript.sh`) para assegurar que nenhum tipo `any` implícito foi adicionado no serviço de pedidos.
- Quando o `/goal` longo de reestruturação é concluído com sucesso e a branch do Git Worktree está pronta, o hook de finalização de tarefa dispara uma notificação física para o desenvolvedor: um aviso sonoro no terminal (beep) e um pop-up visual no sistema operacional. Isso evita que você precise policiar o terminal em tarefas demoradas, permitindo que você retorne à cabine assim que o trabalho agêntico for finalizado.

```
# Exemplo de script de notificação física acionado por Hook pós-sucesso
osascript -e 'display notification "CRUD de Pedidos implementado e testado com sucesso no Git Worktree!" with
title "Claude Code"'
echo -e "\a" # Emite um sinal sonoro (beep) no terminal host
```

O Ecossistema da Comunidade e a Economia de Tokens

O desenvolvimento AI-native moderno não se limita às ferramentas oficiais da Anthropic. Existe uma comunidade vibrante que cria ferramentas fundamentais para otimização de custos, gerenciamento de memória e indexação de bases de código:

1. **Economia de Tokens e Proxy de Linha de Comando**:
    - **RTK (Rust Token Killer)** (`github.com/rtk-ai/rtk`): Um proxy otimizado escrito em Rust que intercepta e comprime logs de terminal, outputs de linters e testes gigantescos antes de enviá-los ao LLM, gerando economias drásticas de 60% a 90% em operações diárias de desenvolvimento.
    - **Caveman** (`github.com/JuliusBrussee/caveman`): Uma alternativa focada no controle rígido de consumo de tokens em ambientes CLI locais.
2. **Repositórios de Skills e Plugins**:
    - **aitmpl.com/skills/** e **skills.sh**: Marketplaces públicos e repositórios comunitários para descobrir, compartilhar e baixar skills prontas de auditoria, refatoração de código, testes e deploys.
    - **context-7** (context7.com (https://context7.com)): Plataforma de documentação em tempo real que alimenta agentes (como Cursor e Claude Code) com referências atualizadas e exemplos práticos via CLI

(`npx ctx7 setup`) ou servidores MCP para evitar alucinações de APIs obsoletas.
    - **frontend-design** (github.com/anthropic/frontend-design-skill (https://github.com/anthropic/frontend-design-skill)): Skill oficial de curadoria visual da Anthropic, direcionando o LLM a adotar estéticas profissionais premium, banir clichês (“AI slop”) e fontes padrão de navegador.
    - **ui-ux-pro-max** (github.com/nextlevelbuilder/ui-ux-pro-max-skill (https://github.com/nextlevelbuilder/ui-ux-pro-max-skill)): Base de conhecimento e gerador inteligente de sistemas de design, cobrindo mais de 50 estilos visuais, paletas curadas e guias de acessibilidade para Next.js, React e SwiftUI.
3. **Gerenciamento de Memória Persistente**:
    - **claude-mem** (`github.com/thedotmack/claude-mem`): Um utilitário de persistência de memória a longo prazo para o Claude Code que estende as capacidades nativas do agente de recordar preferências, padrões de arquitetura e decisões de sessões passadas entre reinicializações de máquina.
4. **Indexação com Grafos de Dependência**:
    - **lemon-code-graph** (`github.com/Andersonlimahw/lemon-code-graph`): Ferramenta que gera grafos de chamadas e mapeamentos de arquivos no repositório local, otimizando o envio de dependências cruzadas para a janela de contexto do agente.
    - **Open Graph** (`github.com/colbymchenry/codegraph`): Analisador de grafos de código para contextualização agêntica aprofundada.
5. **LLM-Wiki de Karpathy**:
    - O guia definitivo e minimalista de referência de arquitetura mental de LLMs mantido por Andrej Karpathy: LLM-Wiki Gist (https://gist.github.com/karpathy/442a6bf555914893e9891c11519de94f).

Ferramentas e CLIs Equivalentes ao Claude Code

Para além do Claude Code, o cenário de desenvolvimento AI-native conta com diversas alternativas e proxies equivalentes focados em comandos via terminal e automação nativa:

Empresa	CLI / Ferramenta	Link
Anthropic	Claude Code	code.claude.com (https://code.claude.com)
Google DeepMind	Antigravity CLI	deepmind.google/antigravity (https://deepmind.google)
Google	Gemini CLI	ai.google.dev/gemini-api (https://ai.google.dev)
OpenAI / Codex	Codex CLI	openai.com/blog/openai-codex (https://openai.com/blog/openai-codex)
OpenCode	OpenCode CLI	opencode.dev (https://opencode.dev)
Anysphere	Cursor CLI	cursor.com (https://cursor.com)

Síntese Prática: O Modelo Mental Cristalizado

Se você precisar reter apenas a essência deste e-book para guiar seu fluxo de trabalho amanhã:

> **LLM** é o cérebro. **Harness** é o corpo físico e executor. **Agent** é o especialista contextualizado por arquivos markdown. **Subagents** representam a squad operando em sandbox. **Context** é a memória de trabalho (RAM) limpa e curada. **Skills** são as competências declarativas que o agente aprende sob demanda. **MCP** representa os canais e pontes estáveis para o mundo externo. **Plugins** são os pacotes de distribuição e compartilhamento desse ecossistema. E o **CLI** é a sua cabine de controle direto no host.

Por onde começar amanhã

Teoria sem aplicação prática é rapidamente esquecida. Comece a transformar seu fluxo de trabalho em AI-native seguindo estes passos incrementais:

1. **Crie seu primeiro Agent customizado**: Mapeie uma tarefa puramente sua (ex: criar changelogs com base em commits do git ou rodar migrations de banco) e escreva um arquivo `.claude/agents/changelog-builder.md` com `name`, `description` e regras estritas de atuação.
2. **Defina um Custom Command**: Crie uma automação em `.claude/commands/deploy-check.md` para rodar linters, testar tipos e gerar um relatório unificado de saúde do código antes de commits importantes.
3. **Monitore e Economize**: Adicione o proxy `rtk` na frente do seu terminal e monitore os ganhos de tokens diários ao rodar sessões repetidas.
4. **Utilize Git Worktrees**: Nunca mais trave sua branch principal com sessões longas de agentes `/goal`. Crie um diretório temporário conectado ao git e deixe o agente trabalhar isoladamente enquanto você continua desenvolvendo em paralelo.

O stack está montado e ao seu alcance. A cabine de comando do terminal está aberta. É hora de projetar e operar com a sua própria squad de engenharia agêntica.

Fontes de Referência e Estudo Contextual

- **Anthropic Agent SDK**: Referência de design para o ciclo de vida do Agent Loop: https://code.claude.com/docs/pt/agent-sdk/agent-loop (https://code.claude.com/docs/pt/agent-sdk/agent-loop)
- **Claude Code CLI & Worktrees**: Melhores práticas e isolamento de processos agênticos: https://code.claude.com/docs/pt/worktrees (https://code.claude.com/docs/pt/worktrees)
- **Best Practices Anthropic**: Recomendações oficiais de uso de ferramentas locais no host: https://code.claude.com/docs/pt/best-practices (https://code.claude.com/docs/pt/best-practices)
- **Ecosystem and Memory**: Estudo sobre o comportamento de engenharia de contexto e memória: https://code.claude.com/docs/pt/memory (https://code.claude.com/docs/pt/memory)
- **Lemon Blog**: Para aprofundar nos fundamentos de agentes e como escrever instruções ricas, veja o artigo: Como criar agents customizados com Claude Code (https://lemon.dev.br/pt/blog/o-que-e-um-agent-e-como-criar-agents-customizados-com-claude-code)

- **Lemon Blog**: Para entender mais sobre otimização do ciclo de desenvolvimento com ferramentas de economia de tokens, leia: Entendendo a economia de tokens em CLI agênticos (https://lemon.dev.br/pt/blog/economia-tokens-stack-skills-cli)
- **Lemon Blog**: Para compreender o funcionamento interno e a arquitetura por trás da correção recursiva de erros, confira: Implementando loops recursivos de correção em AI Harness (https://lemon.dev.br/pt/blog/skill-recursiva-feedback-loop-harness)

Voltar ao índice.

Capítulo 11 — Embeddings & Semantic Search

Como modelos transformam texto em vetores semânticos e como o agente busca por significado, não por palavra.

> Um embedding é a tradução de um texto em um ponto no espaço. Textos com significados próximos viram pontos próximos — e “próximo” passa a ser uma conta de geometria, não uma busca por palavra exata.

TL;DR: Embeddings convertem texto em vetores; buscar por similaridade entre esses vetores é buscar por significado. É a fundação técnica do RAG, da memória e de quase toda recuperação que alimenta um agente.

Aqui começa a **Parte II — AI Native em Produção**. Até o Capítulo 10 o fio condutor foi um CRUD de Pedidos — um exemplo de ensino, deliberadamente simples. Agora subimos o nível: o caso de estudo passa a ser um produto real rodando em produção, a **IgnitionStack** (ignitionstack.pro (https://www.ignitionstack.pro/pt)) — uma plataforma SaaS multi-tenant (autenticação, billing com Stripe, workspaces, convites, dashboard) onde a IA não é demo, é parte do fluxo de desenvolvimento. O domínio `order` dos capítulos anteriores é só uma das features que vivem dentro dela.

Primeiro, o embedding em ação

Um desenvolvedor da IgnitionStack abre o assistente interno e digita:

```
> como faço pra cobrar por assento em vez de por workspace?
```

A documentação interna nunca usou a palavra “assento”. O termo lá dentro é **seat-based billing**, e o trecho relevante fala em “cobrança por usuário ativo no workspace”. Uma busca por palavra-chave (`LIKE '%assento%'`) retornaria **zero resultados**. Mesmo assim, o assistente devolve o documento certo:

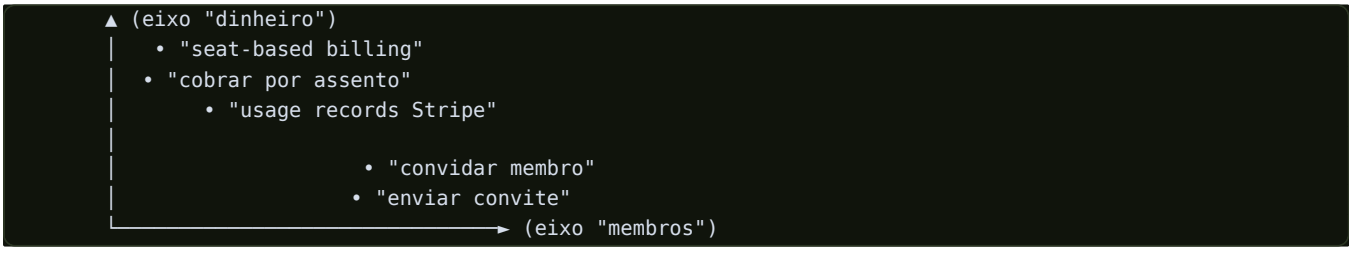
```
[busca semântica]
  consulta: "cobrar por assento em vez de por workspace"
  → doc 0.89  billing/seat-based.md      "cobrança por usuário ativo..."
  → doc 0.81  billing/stripe-metered.md  "Stripe usage records por seat"
  → doc 0.62  workspaces/limits.md        "limite de membros por plano"
```

O número ao lado de cada documento é a **similaridade**. O sistema não procurou a palavra — procurou o *significado*. “Assento”, “seat” e “usuário ativo” caem perto no espaço vetorial porque o modelo aprendeu que descrevem a mesma ideia. Essa é a mágica que vamos abrir.

O que é um embedding

> Um **embedding** é um vetor de números (tipicamente 256 a 3072 dimensões) que representa o significado de um texto. Um modelo de embedding é treinado para que textos semanticamente próximos produzam vetores próximos no espaço.

Pense em cada texto como um ponto num espaço de centenas de dimensões. Você não consegue visualizar 1024 eixos, mas a intuição em 2D vale:



“Cobrar por assento” e “seat-based billing” ficam colados; “enviar convite” fica num bairro diferente. O modelo de embedding aprendeu essa geometria lendo bilhões de textos. Você não programa as regras — você herda o significado destilado no treino.

Como se mede “próximo”: cosine similarity

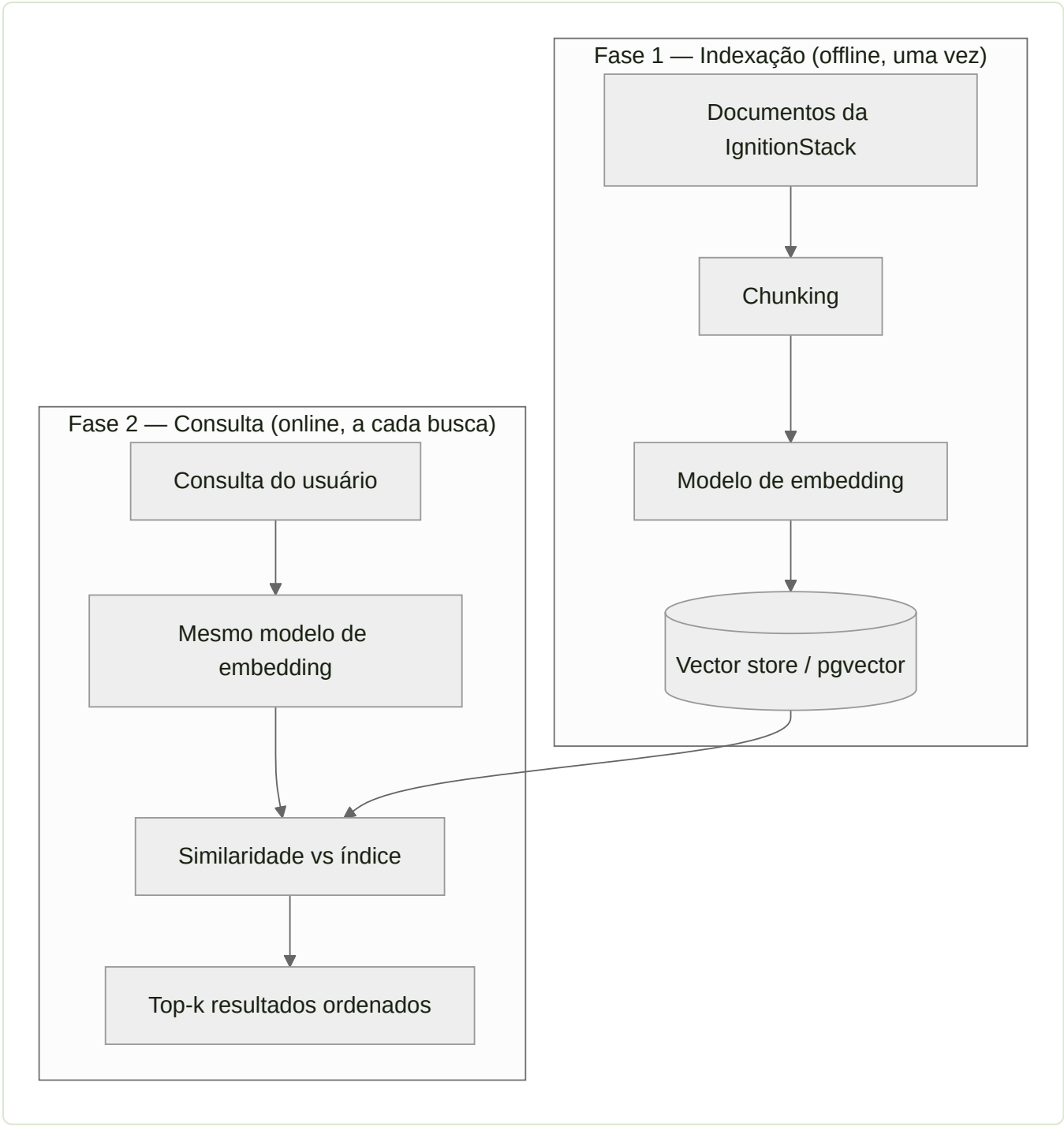
A métrica padrão é a **similaridade do cosseno** — o cosseno do ângulo entre dois vetores:

```
cosine(A, B) = (A · B) / (‖A‖ · ‖B‖)

  =  1  → mesma direção  (significado quase idêntico)
  =  0  → perpendicular  (sem relação)
  = -1  → oposto
```

Usamos o ângulo, não a distância em linha reta (euclidiana), porque o que importa é a **direção** do significado, não a magnitude do vetor. A maioria das APIs já devolve vetores normalizados (‖A‖ = 1), então o cosseno vira um simples produto escalar — barato de calcular para milhões de vetores.

Como a busca semântica funciona por dentro

Há duas fases distintas, e confundi-las é a fonte de metade dos bugs:



Três decisões definem a qualidade:

1. **Chunking** — você não indexa um arquivo de 4.000 linhas inteiro. Quebra em pedaços (*chunks*) de algumas centenas de tokens. Chunk grande demais dilui o sinal (um vetor tenta representar cinco assuntos); pequeno demais perde contexto (um parágrafo sem o título que o explica). Estratégias comuns: tamanho fixo com *overlap* (sobreposição de ~10-15% para não cortar uma ideia ao meio), quebra por estrutura (por `##` no Markdown) ou *semantic chunking* (corta onde o assunto muda).

2. **Mesmo modelo nas duas fases** — o vetor da consulta e o vetor dos documentos **precisam** vir do mesmo modelo. Trocar o modelo de embedding invalida o índice inteiro: você terá de reindexar tudo. Isso tem custo real, e voltaremos a ele.
3. **Nearest neighbors** — achar os k vizinhos mais próximos. Comparar a consulta com *todos* os vetores (busca exata, *brute force*) é O(n) e não escala para milhões. Por isso usamos **ANN** (Approximate Nearest Neighbors) — índices como **HNSW** (grafo navegável) ou **IVF** (partições) que trocam um pouco de precisão por uma busca ordens de magnitude mais rápida.

Na prática: indexar e buscar com pgvector

A IgnitionStack já roda Postgres. Em vez de adicionar um banco vetorial dedicado, ela usa a extensão `pgvector` — menos infraestrutura, transações junto com os dados de negócio:

```sql
-- Indexação: uma coluna de vetor + índice ANN
CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE doc_chunks (
  id          bigserial PRIMARY KEY,
  tenant_id   uuid NOT NULL,          -- multi-tenancy: isolamento por tenant
  source      text NOT NULL,
  content     text NOT NULL,
  embedding   vector(1024)            -- dimensão do modelo escolhido
);

-- HNSW: busca aproximada rápida, ótima para leitura intensiva
CREATE INDEX ON doc_chunks USING hnsw (embedding vector_cosine_ops);
```

```ts
// Consulta: embeddar a pergunta e buscar os k vizinhos — sempre com filtro de tenant
async function semanticSearch(tenantId: string, query: string, k = 5) {
  const [vector] = await embed([query]); // mesmo modelo da indexação

  // <=> é o operador de distância de cosseno do pgvector (menor = mais próximo)
  return sql`
    SELECT source, content, 1 - (embedding <=> ${vector}::vector) AS score
    FROM doc_chunks
    WHERE tenant_id = ${tenantId}
    ORDER BY embedding <=> ${vector}::vector
    LIMIT ${k}
  `;
}
```

Repare no `WHERE tenant_id` antes da ordenação por similaridade: num SaaS multi-tenant, **vazar o chunk de um tenant para outro é um incidente de segurança**, não um bug de relevância. O filtro de isolamento vem primeiro; a busca semântica opera dentro dele.

Quando embeddings funcionam — e quando falham

Embeddings brilham em **significado difuso** e tropeiam em **precisão literal**. Saber a diferença evita escolher a ferramenta errada:

Funciona bem	Falha ou atrapalha
Sinônimos e paráfrases (“assento” ↔ “seat”)	Identificadores exatos (`ERR_BILLING_4012` , um SKU, um UUID)
Perguntas em linguagem natural	Negação (“workspaces **sem** billing” pode vir perto de “com billing”)
Conceitos relacionados	Números e datas precisas
Cross-lingual (PT ↔ EN) com bons modelos	Acrônimos ambíguos fora de contexto

A lição de produção: **busca semântica e busca por palavra-chave são complementares, não rivais.** Para um código de erro, um `WHERE code = ...` continua imbatível. Sistemas maduros fazem *hybrid search* — combinam o ranking lexical (BM25) com o vetorial — e é por isso que o próximo capítulo, sobre RAG, trata reranking como peça de primeira classe.

Custos e trade-offs

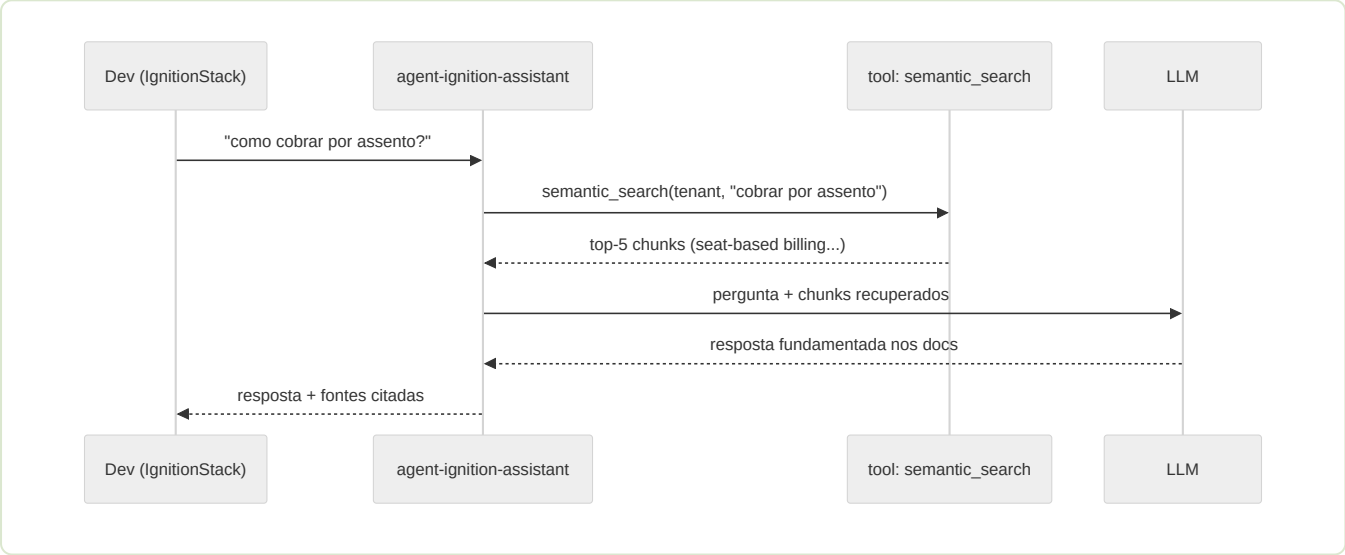
Embeddings parecem grátis até a conta chegar. Os eixos de custo:

- **Geração** — você paga por token embeddado, na indexação e em *toda* consulta. É barato por unidade (ordens de magnitude abaixo da geração de texto), mas reindexar 10 milhões de chunks não é de graça.
- **Armazenamento** — um vetor `float32` de 1024 dimensões ocupa ~4 KB. Multiplique por milhões de chunks e o índice supera os dados originais. *Quantização* (guardar em `int8` ou binário) corta isso a 1/4 ou menos, com perda pequena de precisão.
- **Dimensionalidade** — mais dimensões captam mais nuance, mas custam mais armazenamento, mais memória no índice e buscas mais lentas. Modelos modernos suportam *Matryoshka* (truncar o vetor para menos dimensões com degradação suave) — você escolhe o ponto na curva.

- **A bomba-relógio da reindexação** — trocar de modelo de embedding obriga a recomputar **todo** o índice. Versione qual modelo gerou cada vetor; trate uma migração de modelo como migração de banco.

Conectando ao Agent

Volte ao Capítulo 05: o context é finito e "o que não entra na janela não existe para o modelo". Embeddings são o mecanismo que decide **o que entra**. O agente não carrega a base inteira — ele usa a busca semântica como uma *ferramenta* (no sentido do Capítulo 02): formula uma consulta, recupera os k chunks mais relevantes e injeta só eles no contexto antes de raciocinar.



É a mesma curadoria de contexto do Capítulo 05, agora automatizada por geometria. O `tools: [semantic_search]` no frontmatter de um agent é o que lhe dá memória de longo prazo sem estourar a janela. Esse padrão — recuperar antes de responder — tem nome próprio e é o assunto do próximo capítulo.

Trade-offs e armadilhas

- **Chunk errado, busca errada.** A maior parte das falhas de relevância nasce no chunking, não no modelo. Comece com chunks de ~300-500 tokens e overlap, e ajuste medindo.
- **Dois modelos, índice quebrado.** Embeddar consulta e documentos com modelos diferentes produz lixo silencioso — sem erro, só resultados ruins. Centralize a função `embed()`.

- **Semântica não é exatidão.** Não use embedding para buscar um UUID ou um valor de fatura. Para isso existe o `WHERE`.
- **Multi-tenancy não é opcional no filtro.** Sempre isole por `tenant_id` *antes* da similaridade. Relevância nunca justifica vazamento de dados.
- **Esquecer a freshness.** Um índice é uma foto do passado. Documento mudou e não foi reindexado? O agente responde com a versão velha, confiante. Reindexação incremental é parte do design, não um extra.

Como saber se você entendeu

Você dominou este capítulo se consegue:

- explicar por que “assento” encontra “seat-based billing” sem nenhuma palavra em comum;
- justificar por que consulta e documentos exigem o mesmo modelo de embedding;
- decidir, para um caso dado, entre busca semântica, busca por palavra-chave ou híbrida.

Fontes

- OpenAI — “New embedding models and API updates” (dimensões, Matryoshka, custo): https://openai.com/index/new-embedding-models-and-api-updates/ (https://openai.com/index/new-embedding-models-and-api-updates/)
- pgvector — extensão de vetores para Postgres (HNSW, IVFFlat, operadores): https://github.com/pgvector/pgvector (https://github.com/pgvector/pgvector)
- Malkov & Yashunin, “Efficient and robust approximate nearest neighbor search using HNSW” (2018): https://arxiv.org/abs/1603.09320 (https://arxiv.org/abs/1603.09320)
- Anthropic — “Effective context engineering for AI agents” (recuperação como curadoria de contexto): https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents (https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents)

Síntese

Um embedding transforma significado em geometria: textos próximos viram vetores próximos, e buscar por cosseno é buscar por sentido. Com chunking cuidadoso, o mesmo modelo nas duas pontas e um índice ANN, a IgnitionStack acha o documento certo mesmo quando o usuário usa outra palavra. Mas recuperar é só o primeiro passo — o agente ainda precisa *usar* o que recuperou para responder sem alucinar.

Próximo: Capítulo 12 — RAG.

Capítulo 12 — RAG (Retrieval Augmented Generation)

Como agentes usam conhecimento externo para responder com fatos atuais e citáveis, em vez de alucinar.

> RAG é dar ao modelo a fonte antes de pedir a resposta. Em vez de confiar no que ele “lembra” do treino, você recupera o trecho certo e manda junto com a pergunta.

TL;DR: RAG (Retrieval Augmented Generation) recupera conhecimento externo relevante e o injeta no contexto antes da geração. Resolve três problemas que o modelo sozinho não resolve: conhecimento privado, conhecimento atual e respostas citáveis.

No capítulo anterior demos ao agente a capacidade de *encontrar* o documento certo por significado. RAG é o que ele faz com esse documento: usá-lo como base de verdade para gerar a resposta. É a diferença entre um modelo que *acha* que sabe e um agente que *mostra de onde tirou*.

Primeiro, o RAG em ação

Um cliente da IgnitionStack abre o chat de suporte:

```
> qual o limite de workspaces no plano Pro?
```

Esse número não está nos pesos do modelo — ele mudou na semana passada, é específico da IgnitionStack, e varia por plano. Um LLM puro faria a pior coisa possível: **inventaria um número plausível**. Com RAG, o fluxo é outro:

```
[1. retrieval]  busca "limite de workspaces plano Pro"
   → billing/plans.md#pro   "Pro: até 10 workspaces, 25 seats"   (0.91)
   → billing/limits.md      "limites são soft, com overage..."    (0.74)

[2. augment]    monta o prompt:
   "Contexto: <plans.md#pro + limits.md>. Pergunta: qual o limite..."

[3. generate]   LLM responde SÓ com base no contexto:
   "No plano Pro são até 10 workspaces (e 25 seats). Acima disso há
    cobrança de overage. Fonte: billing/plans.md."
```

Três etapas: **recuperar, aumentar, gerar**. O modelo deixou de ser a fonte da verdade e virou o *redator* de uma verdade que você forneceu. E entregou a fonte junto — auditável.

O que é RAG

> **RAG** é um padrão que, antes de gerar uma resposta, recupera trechos relevantes de uma base de conhecimento externa e os adiciona ao contexto do modelo. A geração fica *ancorada* (grounded) nesses trechos, não apenas na memória paramétrica do treino.

A motivação é direta. Um LLM tem três limitações estruturais que RAG ataca:

- **Não conhece seus dados privados.** Os docs internos, as decisões de arquitetura e o changelog da IgnitionStack nunca estiveram no treino.
- **Está congelado no tempo.** O modelo tem um *cutoff* de conhecimento. O preço que você mudou ontem ele desconhece.
- **Não cita fontes.** Sem RAG, você não tem como saber se a resposta veio de um fato ou de uma alucinação confiante.

”Mas e o contexto infinito?”

A objeção óbvia: se as janelas de contexto crescem (centenas de milhares de tokens), por que não jogar a base inteira no prompt e pular o retrieval? Porque contexto grande não é contexto de graça:

- **Custo.** Você paga por token de entrada em *toda* chamada. Despejar 200k tokens de docs para responder uma pergunta de uma linha é desperdício multiplicado por cada request.

- **Latência.** Quanto mais contexto, mais lenta a resposta. O usuário do suporte não espera 30 segundos.
- **Lost in the middle (Cap. 05).** O modelo aproveita mal a informação enterrada no meio de um contexto gigante. Cinco chunks certos batem cinquenta documentos despejados.
- **Freshness.** Mesmo com janela enorme, alguém precisa decidir *qual* versão dos docs entra. Esse alguém é o retrieval.

Contexto infinito muda o teto, não a disciplina. RAG continua sendo a forma de colocar **o sinal certo** na janela.

Como o pipeline RAG funciona por dentro

RAG de produção raramente é “buscar e responder”. A qualidade vem das etapas entre o retrieval bruto e a geração:

- **Query rewriting.** A pergunta do usuário (“e pra quem tá no plano de cima?”) muitas vezes é vaga ou depende do histórico. Reescrevê-la numa consulta autônoma (“limite de workspaces no plano Enterprise”) melhora o retrieval antes mesmo de buscar.
- **Retrieval híbrido.** Combinar busca vetorial (significado, Cap. 11) com lexical/BM25 (termos exatos como `Pro`, `Enterprise`). Um pega paráfrase, o outro pega o identificador. Juntos cobrem mais.
- **Ranking e reranking.** O retrieval traz, digamos, os 20 candidatos mais próximos — rápido, mas grosseiro. O *reranker* (um *cross-encoder* que lê pergunta e candidato **juntos**) reordena esses 20 e fica com os 5 melhores. É mais caro por par, mas roda sobre poucos candidatos, e é o que mais eleva a precisão final.
- **Geração ancorada.** O prompt instrui explicitamente: “responda **apenas** com base no contexto; se não estiver lá, diga que não sabe”. Isso é o que transforma retrieval em resposta confiável.

Freshness: o índice é uma foto, o produto é um filme

A armadilha silenciosa do RAG é o índice velho. O time da IgnitionStack muda um preço, atualiza um doc, lança um changelog — e o agente continua respondendo com a versão indexada na semana passada, com toda a confiança. Estratégias de produção:

- **Reindexação incremental** disparada por evento (doc salvo → reembedda só aquele chunk), não um *rebuild* noturno do índice inteiro.
- **Metadados de validade** (`updated_at`, `version`) no chunk, para filtrar ou priorizar o mais recente.
- **Invalidção em mudanças críticas** — alterou tabela de preços? O changelog e os docs de billing reindexam na hora.

```
// RAG mínimo de produção: rewrite → retrieve híbrido → rerank → gerar ancorado
async function answer(tenantId: string, question: string) {
  const query = await rewriteQuery(question);          // consulta autônoma
  const candidates = await hybridSearch(tenantId, query, 20); // vetorial + BM25
  const top = await rerank(query, candidates, 5);      // cross-encoder
  const context = top.map((c) => `[${c.source}] ${c.content}`).join("\n\n");

  return generate({
    system:
      "Responda apenas com base no CONTEXTO. " +
      "Se a resposta não estiver nele, diga que não encontrou. Cite a fonte.",
    user: `CONTEXTO:\n${context}\n\nPERGUNTA: ${question}`,
  });
}
```

RAG vs Fine-Tuning vs Prompt Engineering

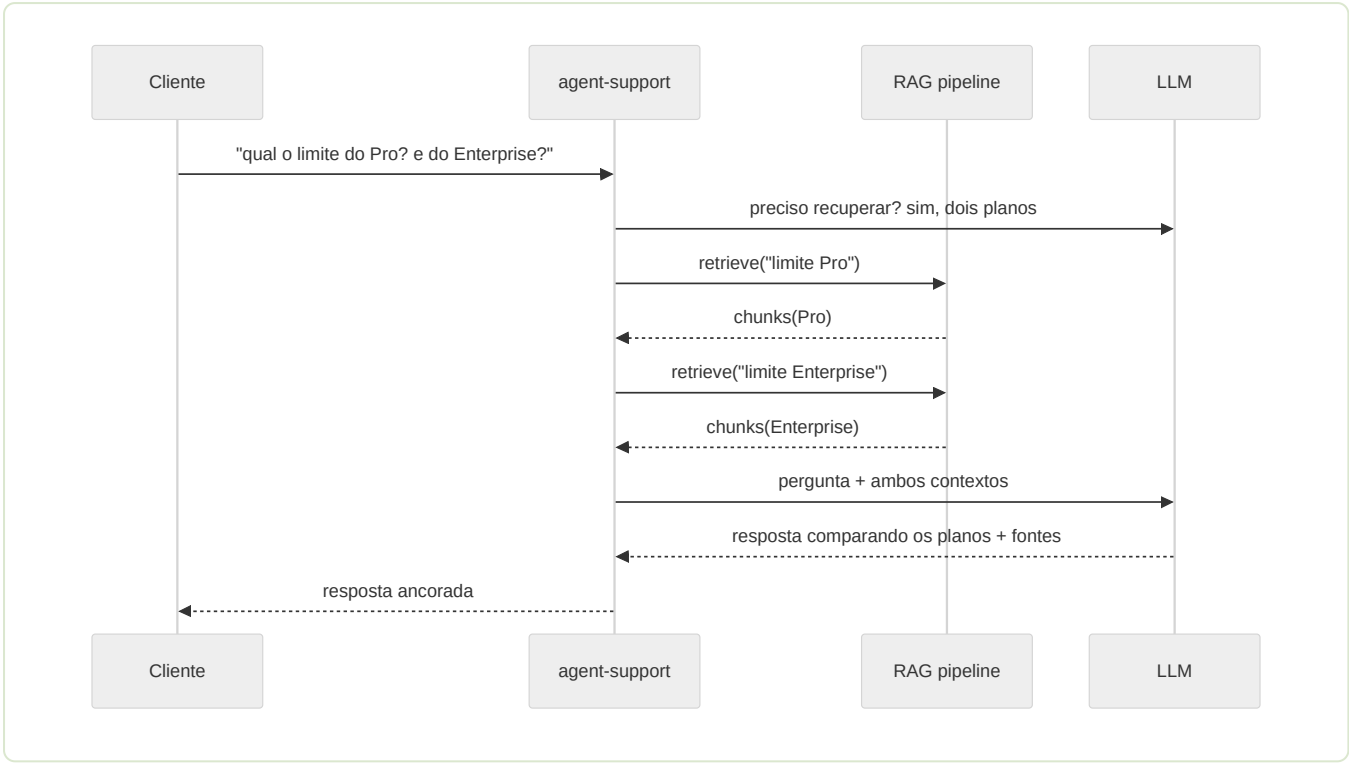
Estas três técnicas resolvem problemas diferentes e são frequentemente confundidas. A regra prática: **prompt primeiro, RAG para conhecimento, fine-tuning para comportamento.**

Critério	Prompt Engineering	RAG	Fine-Tuning
O que ajusta	A instrução	O conhecimento disponível	Os pesos do modelo
Bom para	Formato, tom, raciocínio	Fatos privados e atuais	Estilo/comportamento consistente
Atualização	Instantânea (edita texto)	Reindexar (minutos)	Re-treinar (horas/dias + custo)
Cita fontes	Não	**Sim**	Não
Custo de mudança	~Zero	Baixo	Alto
Risco	Frágil a variações	Retrieval ruim → resposta ruim	*Overfitting*, conhecimento congelado

Para a IgnitionStack, “qual o limite do plano Pro” é trabalho de **RAG** (fato que muda). “Responda sempre em tom de suporte calmo e em PT-BR” é trabalho de **prompt** — e, se precisar ser absolutamente consistente em escala, de **fine-tuning**. Não são rivais; bons sistemas usam as três em camadas.

Conectando ao Agent

No Capítulo 11, o retrieval era uma ferramenta que o agente chamava. RAG eleva isso a um *padrão de raciocínio*: o agente decide **se**, **o que** e **quando** recuperar — o chamado *agentic RAG*.



A diferença para um RAG “burro” (sempre busca uma vez): o agente pode **decompor** a pergunta, recuperar em rodadas, perceber que o retrieval voltou fraco e reformular a consulta. Isso reaproveita exatamente o loop do harness (Cap. 02) — pensar → usar ferramenta → observar → pensar de novo — só que a ferramenta é o retrieval. RAG, no fim, é context engineering (Cap. 05) com uma fonte externa.

Trade-offs e armadilhas

- **Garbage in, garbage out.** RAG não conserta uma base ruim. Doc desatualizado ou errado indexado vira resposta errada com aparência de fonte confiável.
- **Reranking é o maior ganho por real gasto.** Pular o reranker é o erro mais comum; o retrieval bruto traz o relevante *e* o quase-relevante misturados.
- **“Responda só pelo contexto” precisa estar no prompt.** Sem essa âncora explícita, o modelo volta a misturar memória paramétrica e alucina por baixo do RAG.
- **Citações importam.** Sem fonte, você não consegue auditar nem o usuário consegue confiar. Faça o agente citar o `source` de cada afirmação.
- **Freshness é uma feature, não um cron.** Trate reindexação como parte do fluxo de publicação de conteúdo, não como manutenção esquecida.

Como saber se você entendeu

Você dominou este capítulo se consegue:

- explicar por que contexto infinito não substitui retrieval;
- ordenar as etapas rewrite → retrieve → rerank → generate e dizer o que cada uma resolve;
- escolher entre prompt, RAG e fine-tuning para um requisito dado.

Fontes

- Lewis et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks” (2020) — o paper que nomeou RAG: https://arxiv.org/abs/2005.11401 (https://arxiv.org/abs/2005.11401)
- Anthropic — “Contextual Retrieval” (melhorando precisão de retrieval com contexto nos chunks): https://www.anthropic.com/news/contextual-retrieval (https://www.anthropic.com/news/contextual-retrieval)
- Cohere — Rerank (cross-encoder para reordenação de resultados): https://docs.cohere.com/docs/rerank-overview (https://docs.cohere.com/docs/rerank-overview)
- Liu et al., “Lost in the Middle” (por que despejar contexto não basta): https://arxiv.org/abs/2307.03172 (https://arxiv.org/abs/2307.03172)

Síntese

RAG é a ponte entre o que o modelo aprendeu e o que a IgnitionStack precisa que ele saiba *agora*: recupera o trecho certo, ancora a geração nele e devolve a resposta com fonte. Com query rewriting, retrieval híbrido e reranking, a precisão sobe; com reindexação incremental, a resposta não envelhece. Mas tanto embeddings quanto RAG são memória de *fatos*. Falta a memória do *relacionamento* — o que o agente lembra de você, deste tenant, desta conversa.

Próximo: Capítulo 13 — Memory.

Capítulo 13 — Memory

Como sistemas de IA lembram entre chamadas e sessões — e o que devem deliberadamente esquecer.

> O modelo não tem memória. Cada chamada começa do zero. Tudo que parece “lembrança” é, na verdade, alguém recolocando a informação certa de volta na janela de contexto.

TL;DR: Memory é a disciplina de persistir o que importa entre chamadas e sessões — e descartar o resto. Sem ela, todo agente é amnésico; com ela mal feita, vira um risco de privacidade e um buraco de custo.

Embeddings e RAG (Caps. 11-12) deram ao agente memória de *fatos do mundo*. Falta a memória do *relacionamento*: o que este tenant configurou, o que este usuário prefere, o que ficou decidido na conversa de ontem. É isso que transforma uma ferramenta que responde perguntas num assistente que conhece você.

Primeiro, a memória em ação

Compare duas sessões com o assistente de desenvolvimento da IgnitionStack, com uma semana de intervalo.

Sem memória — toda sessão recomeça do zero:

```
Segunda:
> gere o controller de billing
< Em qual linguagem? Qual framework? Como vocês tratam multi-tenancy?

Sexta (mesma pessoa, mesmo projeto):
> gere o controller de convites
< Em qual linguagem? Qual framework? Como vocês tratam multi-tenancy?
```

Com memória — o que foi estabelecido persiste:

```
Segunda:
> gere o controller de billing
< [recupera memória do projeto: TS + Bun.serve, multi-tenant por tenant_id,
   erros via Result<T>]
  Pronto, seguindo o padrão do tenant_id e Result<T> do projeto.

Sexta:
> gere o controller de convites
< Mesmo padrão de billing (tenant_id, Result<T>). Aqui está.
```

A diferença não é inteligência — é **persistência**. Na segunda interação o agente não perguntou de novo porque o fato "este projeto usa TS, Bun e tenant_id" foi *lembrado*. A pergunta de engenharia é: lembrado **onde**, por **quanto tempo**, e **com qual consentimento**.

O que é memória em sistemas de IA

> **Memória** é qualquer mecanismo que reintroduz informação relevante no contexto do modelo ao longo do tempo, já que o modelo em si é *stateless* entre chamadas.

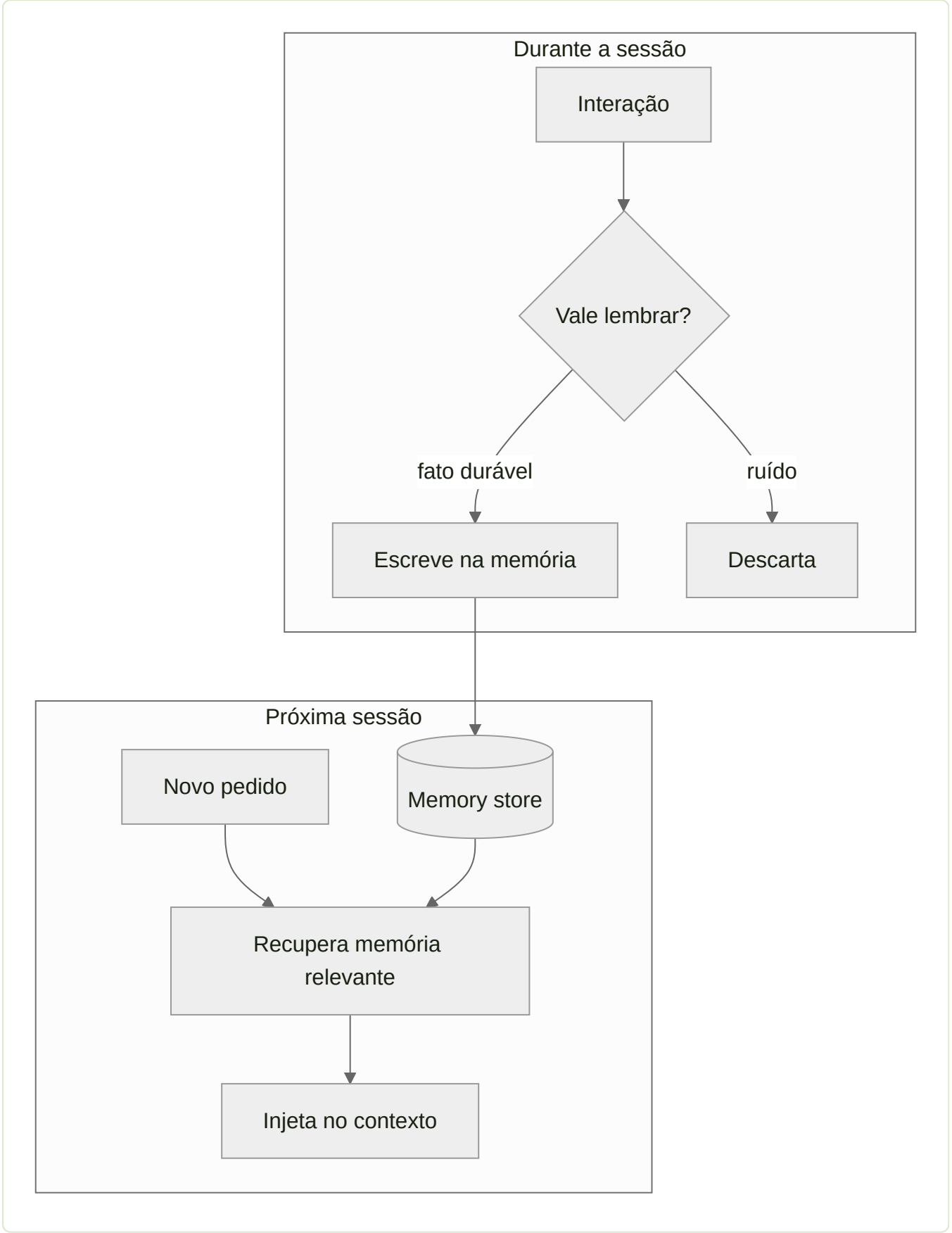
A memória de trabalho do Capítulo 05 (a janela) é volátil: acabou a chamada, sumiu. Memória, neste capítulo, é o que sobrevive a isso. Ela se divide em tipos com propósitos distintos:

Tipo	O que guarda	Exemplo na IgnitionStack	Tempo de vida
Short-term	A conversa atual	o que você pediu há 3 mensagens	a sessão
Long-term	Fatos persistentes	"este projeto usa Bun + tenant_id"	indefinido
Episodic	Eventos e interações passadas	"na sessão de 02/06 decidimos optimistic locking"	longo, com decay
Semantic	Conhecimento generalizado	"padrões de billing recorrentes deste time"	longo
User memory	Preferências da pessoa	"responde em PT-BR, prefere código comentado"	indefinido

Repare que **episodic** e **semantic** se relacionam como o diário e o aprendizado: a memória episódica guarda *o que aconteceu* (eventos brutos); a semântica destila *o que isso ensina* (padrões). Consolidar episódios em conhecimento semântico — e jogar fora os episódios crus — é como a memória escala sem inchar.

Como a memória funciona por dentro

Memória não é um campo mágico — é um *loop de escrita e leitura* em volta do agente, e a parte difícil não é guardar, é **decidir o que guardar e o que recuperar**.



Três planos de memória que a IgnitionStack mantém separados — porque misturá-los é erro de design e de privacidade:

1. **Memória de usuário** — preferências da pessoa. Atravessa projetos, pertence ao indivíduo. (“prefere PT-BR, código comentado.”)
2. **Memória de projeto/tenant** — convenções e decisões daquele workspace. Atravessa sessões, pertence ao tenant. (“padrão tenant_id, Result, Postgres.”)
3. **Memória do agente** — o que o próprio agente aprendeu a fazer melhor (padrões que funcionam, armadilhas recorrentes).

No Claude Code você já viu o plano de projeto na prática: o `CLAUDE.md` é memória de projeto carregada em toda sessão (Cap. 05), e arquivos de memória dedicados guardam fatos de longo prazo. O mesmo princípio, generalizado, vale para qualquer produto de IA.

O que lembrar e o que esquecer

Memória sem critério de esquecimento é um vazamento — de tokens, de custo e de privacidade. As heurísticas que importam:

- **Salience (relevância).** Guarde decisões e fatos duráveis (“usamos optimistic locking”), não trivialidades transitórias (“rode o teste agora”).
- **Decay (decaimento).** Memórias episódicas perdem peso com o tempo, a menos que reforçadas. O que não é reusado, envelhece e some.
- **Consolidação.** Periodicamente, resuma muitos episódios em poucos fatos semânticos. Dez sessões dizendo “ele sempre quer testes” viram uma preferência (“escreve testes por padrão”).
- **Esquecimento ativo.** Algumas coisas devem ser apagadas *por obrigação*, não por economia — e aqui entra a lei.

```
// Escrita seletiva: nem toda interação merece virar memória
async function maybeRemember(scope: MemoryScope, turn: Turn) {
  const fact = await extractDurableFact(turn); // null se for ruído transitório
  if (!fact) return;

  await memory.write({
    scope,                 // 'user' | 'tenant' | 'agent' — nunca misturados
    tenantId: turn.tenantId,
    content: fact.text,
    embedding: await embed([fact.text]),  // recuperável por significado (Cap. 11)
    salience: fact.salience,
    expiresAt: fact.ttl,                  // decay explícito
    containsPII: fact.pii,                // marca para política de privacidade
  });
}
```

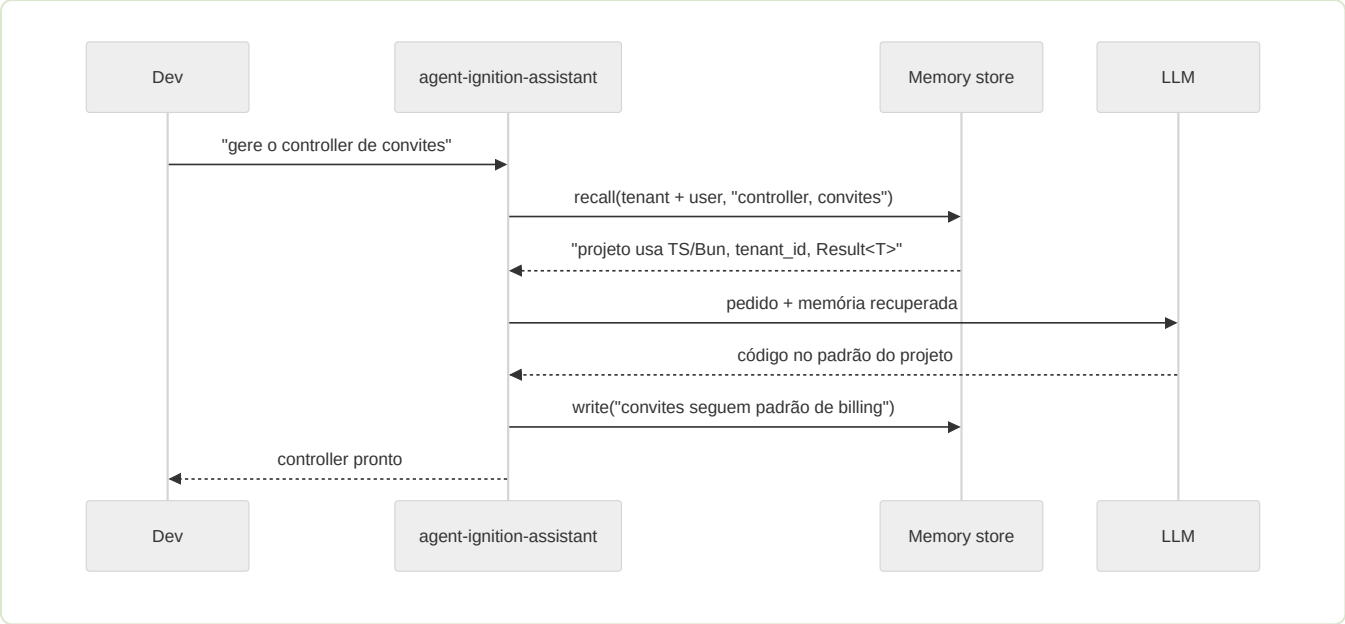
Privacidade, LGPD e custo

Memória é dado pessoal acumulado ao longo do tempo — o terreno exato onde a **LGPD** (Lei Geral de Proteção de Dados) morde. Num SaaS multi-tenant, não é detalhe jurídico, é requisito de arquitetura:

- **Finalidade e consentimento.** Você só pode reter o que tem base legal para reter. "Lembrar para ajudar" não autoriza guardar dado sensível indefinidamente. Memória de usuário precisa de opção de *opt-out*.
- **Direito ao esquecimento.** O usuário pode exigir exclusão. Sua memória precisa de um botão de *delete* real — incluindo os vetores no índice, não só a linha original.
- **Isolamento entre tenants.** Memória do tenant A jamais pode vazar para o B. O `scope` + `tenant_id` do exemplo acima é a fronteira; um recall sem esse filtro é incidente de segurança, igual ao do Capítulo 11.
- **PII no contexto.** Marcar `containsPII` permite redigir ou excluir seletivamente, e evita mandar dado sensível para logs ou para um modelo externo sem necessidade.
- **Custo.** Memória cresce para sempre se você deixar. Cada fato é armazenamento + embedding + tokens de contexto a cada recall. Consolidação e TTL não são só higiene — são controle de custo (assunto do Cap. 17).

Conectando ao Agent

Lembre da frase do Capítulo 05: *"o agent é, em boa parte, uma decisão de context engineering congelada em um arquivo."* Memória é a versão **dinâmica** disso. O system prompt é a memória estática do agente (sempre presente); o memory store é a memória dinâmica (recuperada conforme a relevância).



O loop fecha: o agente **lê** memória antes de agir e **escreve** memória depois de aprender algo durável. É o mesmo mecanismo do RAG (Cap. 12), mas a fonte é o histórico do relacionamento, não a base de documentos. Tecnicamente, memória *é* RAG aplicado à sua própria interação.

Trade-offs e armadilhas

- **Lembrar de tudo é não lembrar de nada.** Memória sem curadoria infla o contexto com ruído e degrada a resposta (lost in the middle, Cap. 05). Escreva seletivamente.
- **Memória errada é pior que ausência.** Um fato obsoleto guardado como verdade (“o projeto usa REST” quando migrou para gRPC) faz o agente errar com confiança. Memória precisa de atualização e invalidação.
- **Misturar escopos vaza dados.** Preferência de usuário ≠ convenção de tenant ≠ aprendizado do agente. Separe na origem; um recall que cruza escopos é bug ou incidente.
- **Esquecer o esquecimento.** Sem TTL e sem delete real, você viola LGPD e acumula custo silenciosamente.
- **Memória implícita engana o usuário.** Se o agente “lembra” algo, deixe rastreável de onde veio — como nas citações do RAG. Memória opaca destrói confiança.

Como saber se você entendeu

Você dominou este capítulo se consegue:

- distinguir short-term, long-term, episodic, semantic e user memory com um exemplo de cada;
- explicar por que consolidação e decay são necessários, não opcionais;
- desenhar como o direito ao esquecimento (LGPD) se implementa num memory store com embeddings.

Fontes

- Anthropic — Memory tool e gerenciamento de memória de agentes: https://docs.anthropic.com/en/docs/build-with-claude/tool-use/memory-tool (https://docs.anthropic.com/en/docs/build-with-claude/tool-use/memory-tool)
- Claude Code — memória (`CLAUDE.md`, memória de projeto e de usuário): https://code.claude.com/docs/en/memory (https://code.claude.com/docs/en/memory)
- Park et al., “Generative Agents: Interactive Simulacra of Human Behavior” (2023) — memória episódica, reflexão e recuperação por relevância: https://arxiv.org/abs/2304.03442 (https://arxiv.org/abs/2304.03442)
- ANPD — Lei Geral de Proteção de Dados (LGPD), texto e guias: https://www.gov.br/anpd/pt-br (https://www.gov.br/anpd/pt-br)

Síntese

O modelo é amnésico; memória é o que reintroduz o passado relevante no presente. Separada por escopo (usuário, tenant, agente), curada por salience e decay, e governada por LGPD, ela transforma uma ferramenta de perguntas num assistente que conhece o contexto do relacionamento. Mas tudo que vimos até aqui — recuperar, lembrar, responder — produz *texto*. Para o agente *agir* no produto (criar um workspace, provisionar um plano), o texto não basta: ele precisa emitir dados estruturados e chamar ferramentas.

Próximo: Capítulo 14 — Structured Outputs & Tool Calling.

Capítulo 14 — Structured Outputs & Tool Calling

Como transformar texto livre do modelo em dados validados e ações reais no sistema, com determinismo.

> Texto livre é ótimo para humanos lerem e péssimo para máquinas executarem. Structured outputs e tool calling são como o modelo deixa de *descrever* uma ação e passa a *disparar* uma.

TL;DR: Structured outputs forçam o modelo a responder num formato validável (JSON Schema); tool calling deixa o modelo invocar funções do seu sistema. Juntos, são a ponte entre a linguagem e a execução — e o ponto onde a confiabilidade de um produto de IA se decide.

Os capítulos anteriores produziram *texto*: respostas, fatos recuperados, memórias. Mas a IgnitionStack não vende texto — ela provisiona workspaces, cria produtos no Stripe, gera código. Para o agente *fazer* essas coisas, a saída precisa ser uma instrução estruturada que o sistema executa sem adivinhar.

Primeiro, o tool calling em ação

Um usuário no onboarding da IgnitionStack digita, em linguagem natural:

```
> cria um projeto SaaS chamado Acme, plano Pro, billing mensal, e já convida
  ana@acme.com como admin
```

Um LLM “normal” responderia com um texto: *“Claro! Para criar o projeto Acme...”* — bonito e **inútil** para automação. Com tool calling, o modelo emite uma chamada de ferramenta estruturada que o sistema executa de fato:

```
{
  "tool": "create_project",
  "input": {
    "name": "Acme",
    "plan": "pro",
    "billing_cycle": "monthly",
    "invites": [{ "email": "ana@acme.com", "role": "admin" }]
  }
}
```

O harness (Cap. 02) recebe esse JSON, **valida** contra um schema, e só então executa: cria o workspace, registra o produto no Stripe, dispara o convite. A linguagem natural virou uma transação. E porque a saída é estruturada, ela é *determinística de consumir*: o código que provisiona não precisa interpretar prosa.

Os três níveis de estrutura

Há um espectro de quão “amarrada” é a saída do modelo, e escolher o nível certo é uma decisão de engenharia:

Nível	O que é	Quando usar	Risco
Texto livre	String sem garantia de forma	resposta para humano ler	impossível de parsear com confiança
JSON estruturado	Saída conforme um JSON Schema	extrair dados, preencher formulário	schema fraco → campos faltando
Tool calling	Modelo escolhe e chama uma função	executar ações, orquestrar	chamar ferramenta errada / demais

A diferença entre os dois últimos: **structured output** garante o *formato* da resposta (você quer um objeto com estes campos); **tool calling** dá ao modelo *agência* para decidir qual função invocar e com quais argumentos. Todo tool call é structured output por baixo (os argumentos seguem um schema), mas nem todo structured output é uma ação.

JSON Schema é o contrato

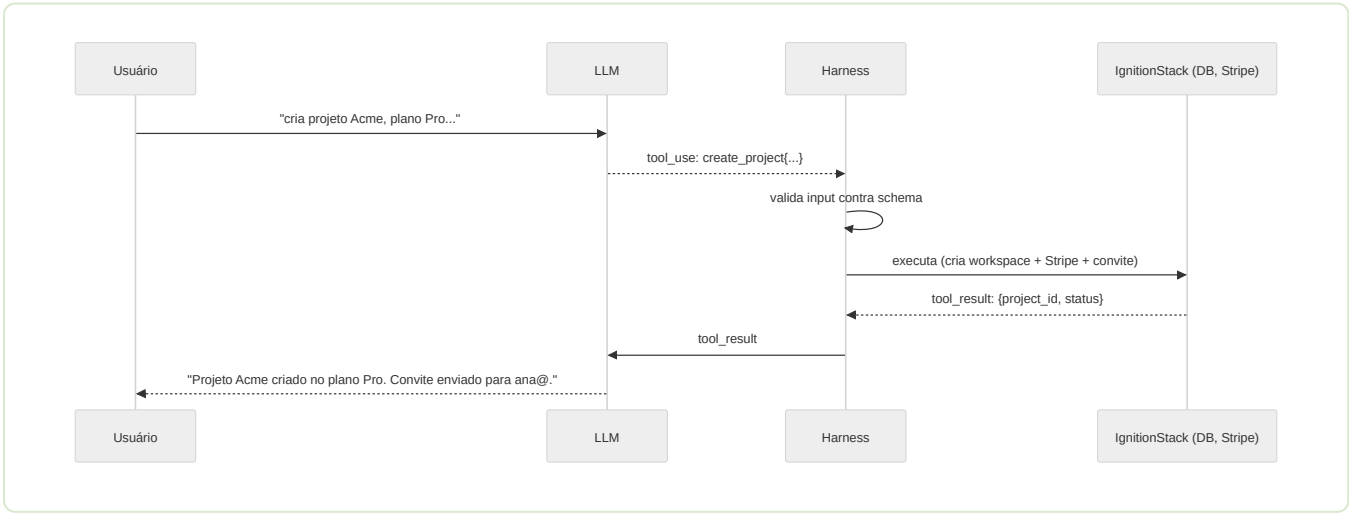
O que torna a saída confiável não é pedir “responda em JSON” — é declarar um **JSON Schema** e fazer o modelo aderir a ele. O schema é o contrato:

```
// O contrato da ferramenta. enums e required são o que evitam alucinação de campo.
const createProjectTool = {
  name: "create_project",
  description:
    "Cria um novo projeto SaaS na IgnitionStack. Use quando o usuário " +
    "pede para criar/provisionar um workspace ou produto.",
  input_schema: {
    type: "object",
    properties: {
      name: { type: "string", minLength: 1, maxLength: 80 },
      plan: { type: "string", enum: ["free", "pro", "enterprise"] }, // fechado!
      billing_cycle: { type: "string", enum: ["monthly", "yearly"] },
      invites: {
        type: "array",
        items: {
          type: "object",
          properties: {
            email: { type: "string", format: "email" },
            role: { type: "string", enum: ["admin", "member", "viewer"] },
          },
          required: ["email", "role"],
        },
      },
    },
    required: ["name", "plan", "billing_cycle"],
  },
} as const;
```

Os `enum` são o detalhe que separa um sistema robusto de um frágil. Sem eles, o modelo pode inventar `plan: "premium"` — um plano que não existe. Com o enum fechado, `"premium"` é impossível de gerar. **Você restringe o espaço de saída no schema, não numa validação depois.**

Como funciona por dentro

Tool calling não é o modelo executando código — é uma negociação em rodadas entre modelo e harness:



Pontos cruciais:

1. **O modelo nunca toca o sistema.** Ele *pede* uma ferramenta; o harness decide se executa. Essa indireção é onde entram permissões, validação e os hooks do Capítulo 07 — sua camada de segurança.
2. **O resultado volta para o modelo.** Depois de executar, o `tool_result` (sucesso, ID gerado, ou erro) retorna ao contexto, e o modelo continua o raciocínio. É o loop do harness (Cap. 02) de novo, agora com ferramentas que mudam o mundo.
3. **Validação é obrigatória, não opcional.** Mesmo com schema, valide o input antes de executar. O modelo erra; o schema reduz a chance, a validação garante a regra.

Determinismo: o que dá e o que não dá para garantir

Aqui está a verdade honesta que muitos produtos ignoram: **o LLM é não-determinístico, mas o consumo da saída pode ser determinístico.** Você não controla *se* o modelo vai chamar `create_project` numa borda ambígua — controla que, *se* chamar, o input estará conforme o schema ou será rejeitado. O determinismo mora na fronteira (schema + validação + idempotência), não no miolo (a geração).

Por isso ações que mudam estado precisam de **idempotência**: se o modelo, por algum motivo, emitir `create_project` duas vezes, uma chave de idempotência evita criar dois workspaces Acme.

```
// A fronteira determinística: valida → idempotência → executa
async function runTool(call: ToolCall, ctx: Ctx) {
  const input = validate(createProjectTool.input_schema, call.input); // lança se inválido
  return withIdempotencyKey(call.id, () => createProject(ctx.tenantId, input));
}
```

Falhas reais e como mitigar

Tool calling falha de formas específicas. As que mais aparecem em produção na IgnitionStack — e a mitigação de cada uma:

Falha	Sintoma	Mitigação
Campo alucinado	`plan: "premium"` (não existe)	`enum` fechado no schema
Argumento faltando	sem `billing_cycle`	`required` + validação que rejeita e pede de novo
JSON malformado	string cortada, vírgula sobrando	structured outputs nativos (constrained decoding), não regex
Over-calling	chama 5 ferramentas quando 1 bastava	`description` precisa de *quando* usar; instrução de parcimônia
Wrong tool	usa `delete_project` em vez de `archive`	descriptions distintas e exemplos; nomes sem ambiguidade
Ação dupla	cria dois workspaces	idempotência por `call.id`

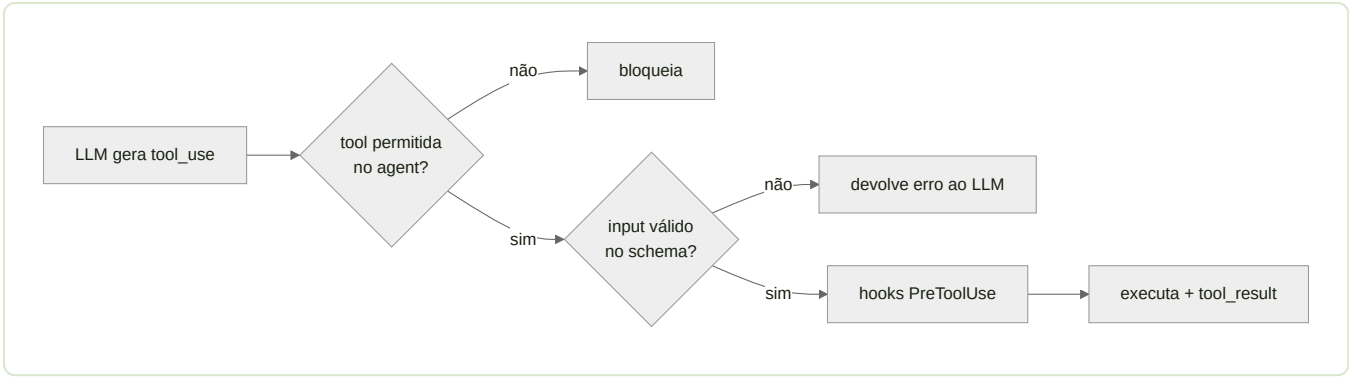
A mitigação transversal: **trate o erro de validação como um turno a mais, não como crash.** Devolva ao modelo “faltou `billing_cycle`, escolha entre monthly/yearly” e deixe-o corrigir. O loop do harness foi feito exatamente para isso.

Conectando ao Harness e ao Agent

Tool calling é onde o agent (Cap. 03) e o harness (Cap. 02) se encontram de forma mais concreta. Releia o frontmatter do agente:

```
tools: Read, Grep, Glob   # o que o agent PODE pedir
```

Esse campo é, precisamente, **a lista de ferramentas que o modelo pode emitir como tool calls**. Restringir `tools` (princípio do menor privilégio do Cap. 03) é restringir quais funções o modelo consegue invocar — não importa o que ele “queira”. O harness é o porteiro: recebe o tool call, confere se está na lista permitida, valida o input contra o schema, aplica hooks, e só então executa.



Em uma frase: **structured outputs dão forma ao que o modelo diz; tool calling dá ação ao que o agent faz; o harness garante que essa ação seja segura, validada e reversível.** É a tríade que transforma um chat num produto.

Trade-offs e armadilhas

- **Schema fraco é a raiz de quase toda falha.** Campos sem `enum`, sem `required`, sem limites — cada folga é uma alucinação possível. Aperte o schema antes de culpar o modelo.
- **Validar não é opcional, mesmo com structured outputs.** O modo nativo reduz JSON inválido, não elimina regra de negócio. Valide sempre na fronteira.
- **Ações que mudam estado exigem idempotência.** Sem ela, um retry vira um efeito colateral duplicado.
- **`description` da ferramenta é tão importante quanto a do agent.** É o que o modelo lê para decidir *quando* chamar. Vago → over-calling ou wrong tool.
- **Não dê `delete`/`charge` de graça.** Ferramentas destrutivas ou que mexem em dinheiro merecem confirmação humana ou hook de aprovação. O Stripe não tem “ctrl+Z”.

Como saber se você entendeu

Você dominou este capítulo se consegue:

- explicar a diferença entre texto livre, structured output e tool calling com um caso de uso de cada;
- mostrar por que `enum` e `required` no schema previnem classes inteiras de falha;
- localizar onde o determinismo é possível (fronteira) e onde não é (geração).

Fontes

- Anthropic — Tool use (definição, `input_schema`, ciclo de tool_use/tool_result): https://docs.anthropic.com/en/docs/build-with-claude/tool-use (https://docs.anthropic.com/en/docs/build-with-claude/tool-use)
- OpenAI — Structured Outputs (constrained decoding, JSON Schema garantido): https://platform.openai.com/docs/guides/structured-outputs (https://platform.openai.com/docs/guides/structured-outputs)
- JSON Schema — especificação oficial (`enum`, `required`, `format`): https://json-schema.org/ (https://json-schema.org/)
- Anthropic — “Building effective agents” (ferramentas como interface agente↔mundo): https://www.anthropic.com/research/building-effective-agents (https://www.anthropic.com/research/building-effective-agents)

Síntese

Structured outputs e tool calling são a fronteira onde a linguagem vira execução: o modelo emite JSON conforme um schema, o harness valida e executa, o resultado volta ao loop. Com `enum`, `required`, validação e idempotência, a IgnitionStack transforma “cria um projeto Acme” numa transação confiável — apesar de o modelo ser, por natureza, não-determinístico. Mas como você sabe que o agente chama a ferramenta *certa* na maioria das vezes? Como medir que a v2 não regrediu? Isso exige medição sistemática.

Próximo: Capítulo 15 — Evals.

Capítulo 15 — Evals

Como medir a qualidade de agentes de forma sistemática e impedir que melhorias em um lugar quebrem outro.

> Você não faz deploy de código sem testes. Por que faria deploy de um agente sem evals? A diferença é que o agente não falha com um stack trace — falha com uma resposta plausível e errada.

TL;DR: Evals são a suíte de testes de sistemas de IA. Medem qualidade contra exemplos de referência, pegam regressões antes do deploy e continuam medindo em produção. Sem eles, “melhorar o prompt” é chute, não engenharia.

Até aqui o agente recupera, lembra e age (Caps. 11-14). Mas como você *sabe* que ele faz isso bem? E quando trocar uma instrução para corrigir um caso, como garante que não quebrou dez outros? Software resolve isso com testes. IA resolve com evals — e a maioria dos times descobre tarde demais que precisava deles.

Primeiro, o eval em ação

O time da IgnitionStack mantém um agente que gera scaffolds de CRUD a partir de uma descrição. Alguém ajusta o prompt para melhorar a geração de validações. Parece uma melhoria inofensiva. Sem evals, vai para produção no escuro. Com evals, isso aparece no CI:

```
$ bun run evals

  scaffold-agent  v43  →  v44 (candidato)

  ✓ compila (tsc)              48/50   (era 49/50)  ▼ -1
  ✓ testes passam              45/50   (era 45/50)  =
  ✓ usa tenant_id              50/50   (era 50/50)  =
  ✓ validação presente         50/50   (era 41/50)  ▲ +9   ← a melhoria
  ✗ sem segredo hardcoded      47/50   (era 50/50)  ▼ -3   ← REGRESSÃO

  RESULTADO: BLOQUEADO — regressão de segurança em 3 casos
```

A melhoria foi real (+9 em validação). Mas a mudança também fez o agente passar a embutir uma API key de exemplo em 3 scaffolds — uma regressão de **segurança** que nenhum humano teria pego revisando “o prompt parece melhor”. O eval

pegou. Isso é a diferença entre achar e medir.

O que são evals

> Um **eval** é um teste automatizado para um sistema de IA: um conjunto de entradas, um critério de qualidade e uma pontuação. Rodado sobre uma versão do agente, produz um número comparável entre versões.

A diferença crucial para um teste unitário: o teste de software é **binário e determinístico** (passou/falhou, sempre igual). O eval lida com saída **não-determinística e gradual** (quão boa foi esta resposta, em média, sobre muitos casos). Por isso evals raramente são “igual a”, e quase sempre “satisfaz um critério” sobre um conjunto.

A escada de evals

Há um espectro do barato-e-objetivo ao caro-e-subjetivo. Sistemas maduros usam todos os degraus, do mais barato pra cima:

Método	Como mede	Custo	Bom para
Verificação programática	código checa o output (`tsc`, testes, regex)	baixíssimo	geração de código, formato, regras objetivas
Golden dataset	compara com resposta de referência	baixo	tarefas com gabarito estável
LLM-as-a-Judge	um modelo pontua o output por rubrica	médio	qualidade subjetiva em escala
Human evaluation	pessoas avaliam	alto	calibrar os métodos acima, casos críticos
Production eval	mede sobre tráfego real	contínuo	o que os outros não preveem

A regra de ouro: **prefira o método mais barato que ainda mede o que importa.** Para “o código compila?”, `tsc` é definitivo e custa quase nada — não chame um LLM-judge para isso. Reserve o judge para o que só semântica resolve (“a resposta foi útil e correta?”).

Como os evals funcionam por dentro

Golden datasets e regression testing

O coração de tudo é um **golden dataset**: casos de entrada com a saída esperada (ou critérios de aceitação). Ele é seu gabarito versionado. Toda vez que o agente muda, você roda contra o mesmo golden e compara.

```
// Um caso de golden dataset para o scaffold-agent
const goldenCase = {
  id: "scaffold-orders-001",
  input: "CRUD de pedidos multi-tenant com transição de status",
  // critérios verificáveis, não uma string exata (a saída é não-determinística)
  checks: [
    { name: "compila", fn: (out) => tscPasses(out) },
    { name: "usa tenant_id", fn: (out) => /tenant_id/.test(out) },
    { name: "sem segredo", fn: (out) => !hasHardcodedSecret(out) },
    { name: "máquina de estados", fn: (out) => hasStateMachine(out) },
  ],
};
```

O **regression testing** é só isto rodado entre versões: a v44 não pode pontuar abaixo da v43 nos checks existentes. Uma melhoria que regride outro critério é **bloqueada** — exatamente como no exemplo da abertura. Sem golden dataset, “melhorou?” é opinião; com ele, é um diff de números.

LLM-as-a-Judge: medir o subjetivo

Para qualidade que código não captura (“a explicação foi clara?”, “a resposta de suporte foi empática e correta?”), usa-se outro modelo como avaliador, guiado por uma **rubrica**:

```
const judgePrompt = `
Avalie a resposta de suporte da IgnitionStack de 1 a 5.
Rubrica:
- Correção factual (a info bate com a documentação fornecida?)
- Completude (respondeu o que foi perguntado?)
- Tom (profissional e claro?)
Responda em JSON: { "score": n, "reasoning": "..." }   // structured output, Cap. 14
`;
```

Cuidados que separam um judge confiável de um teatro de métricas:

- **Rubrica explícita.** “Avalie de 1 a 5” sem critérios produz ruído. Diga *o que* conta.
- **Forneça a referência.** Para correção factual, dê ao judge a fonte (o doc do RAG, Cap. 12). Senão ele julga com a própria alucinação.

- **Structured output (Cap. 14).** Force `{score, reasoning}` para parsear e agregar.
- **Calibre com humanos.** Periodicamente, compare o judge com avaliação humana. Se divergem muito, a rubrica está ruim — não confie cegamente.

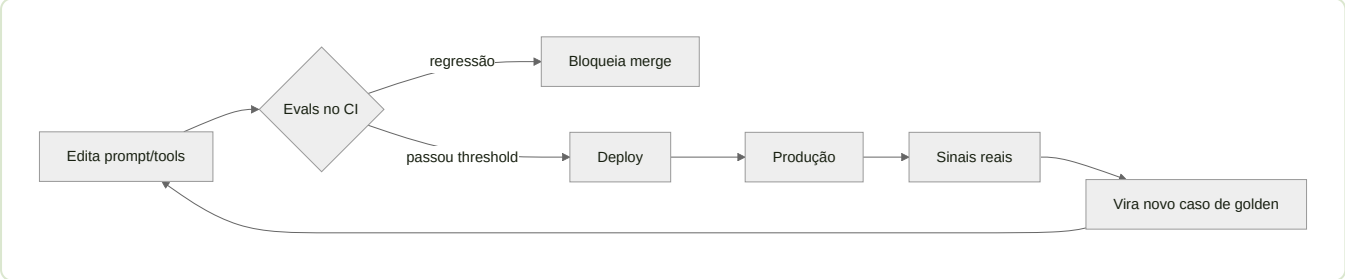
Production eval

O golden dataset cobre o que você *previu*. Produção traz o que você *não* previu. Avaliação em produção mede o agente sobre tráfego real, com sinais como:

- **Implícitos** — o usuário reformulou a pergunta (sinal de resposta ruim)? aceitou o código gerado sem editar (sinal bom)?
- **Explícitos** — 👍/👎, denúncia de resposta errada.
- **Amostragem para judge/humano** — uma fração do tráfego real vira novos casos de golden, fechando o ciclo.

Conectando ao ciclo de vida do Agent

Evals não são uma fase no fim — são um *gate* em cada transição do agente. Releia o agent como software versionável do Capítulo 03: se é software, tem CI. E o CI de um agente são os evals.



O ciclo fecha: produção alimenta o golden dataset, que protege a próxima mudança. Cada bug que escapou vira um caso de eval — então o mesmo erro nunca passa duas vezes. É a disciplina do *test-driven* aplicada ao agente: o eval que falha hoje é a garantia de amanhã.

Onde isso toca as outras camadas: você avalia o **retrieval** (o RAG trouxe o chunk certo? Cap. 12), o **tool calling** (chamou a ferramenta certa com input válido? Cap. 14) e a **resposta final** — cada uma com seu eval. Um agente que erra pode estar errando em qualquer um desses pontos, e só evals por etapa dizem qual.

Trade-offs e armadilhas

- **Sem golden dataset, você não tem evals — tem opinião.** “O prompt parece melhor” não é mensurável. Comece pequeno: 20 casos reais valem mais que zero.
- **LLM-as-a-Judge não é grátis nem infalível.** Custa tokens e tem viés (favorece respostas longas, o próprio estilo). Calibre com humanos; não o use onde código basta.
- **Otimizar para o eval é o novo overfitting.** Se você ajusta o agente até passar nos 20 casos, ele pode ir bem só neles. Mantenha um conjunto de teste que o desenvolvimento não vê.
- **Eval lento não roda.** Se a suíte demora uma hora, ninguém roda antes do merge. Verificação programática barata primeiro; judge só onde precisa.
- **Não medir produção é medir o passado.** O golden envelhece. Sem sinal de produção, você otimiza para casos que já não representam os usuários.

Como saber se você entendeu

Você dominou este capítulo se consegue:

- escolher, para um critério dado, entre verificação programática, golden dataset e LLM-as-a-Judge;
- explicar por que regression testing exige um golden dataset versionado;
- desenhar o ciclo em que sinais de produção viram novos casos de eval.

Fontes

- Anthropic — “Building evals and test cases” (golden datasets, classificadores, judges): https://docs.anthropic.com/en/docs/test-and-evaluate/develop-tests (https://docs.anthropic.com/en/docs/test-and-evaluate/develop-tests)
- OpenAI Evals — framework aberto para avaliação de modelos e agentes: https://github.com/openai/evals (https://github.com/openai/evals)
- Anthropic — “A statistical approach to model evals” (rigor estatístico em avaliação): https://www.anthropic.com/research/statistical-approach-to-model-evals (https://www.anthropic.com/research/statistical-approach-to-model-evals)
- Zheng et al., “Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena” (2023): https://arxiv.org/abs/2306.05685 (https://arxiv.org/abs/2306.05685)

Síntese

Evals são o CI dos agentes: golden datasets pegam regressões, verificação programática mede o objetivo de graça, LLM-as-a-Judge mede o subjetivo em escala, e produção revela o que você não previu. Para a IgnitionStack, é o que impede que melhorar a validação introduza um vazamento de segredo sem ninguém notar. Mas evals dizem *que* algo regrediu — não *por quê* nem *onde* no fluxo de múltiplos passos. Para isso, você precisa enxergar o agente por dentro enquanto ele roda.

Próximo: Capítulo 16 — Observability.

Capítulo 16 — Observability

Como enxergar o que um agente fez, por que decidiu, quanto custou e onde falhou — em produção.

> Quando um agente falha em produção, não há stack trace. Há uma decisão estranha no meio de dez passos. Observabilidade é o que transforma “às vezes ele erra” em “no passo 4 o retrieval voltou vazio”.

TL;DR: Observabilidade agêntica é instrumentar o agente para responder, depois do fato, o que ele fez, por que decidiu, quanto custou e onde falhou. Sem traces, prompts versionados e tracking de custo, todo bug vira adivinhação.

Evals (Cap. 15) dizem *que* o agente regrediu. Observabilidade diz *onde* e *por quê*. São complementares: um mede qualidade antes do deploy; o outro explica comportamento depois. Num sistema de múltiplos passos — recuperar, lembrar, chamar ferramenta, gerar — saber em qual passo a coisa desandou é a diferença entre corrigir em minutos e chutar por dias.

Primeiro, a observabilidade em ação

Um cliente da IgnitionStack reclama: “o assistente respondeu errado sobre meu limite de plano”. Sem observabilidade, a investigação é um interrogatório frustrante. Com um *trace* do agente, é leitura direta:

```
trace_id: 7f3a...  agent: support  tenant: acme  custo: $0.011  latência: 4.2s

├─ [span] query_rewrite        120ms   "limite do meu plano" → "limite workspaces plano Pro"
├─ [span] retrieval            340ms   3 chunks  ⚠ score top=0.48 (baixo!)
├─ [span] rerank               210ms   reordenou, top ainda 0.51
├─ [span] llm.generate        3.4s    in:2.1k out:180 tokens  model: sonnet-4-6
└─ [output] "Pro permite 5 workspaces"   ✗ (correto: 10)
```

O trace conta a história inteira: o retrieval voltou com **score 0.48** — fraco. O agente recuperou o chunk errado (provavelmente uma versão antiga do doc de planos), e o modelo, fiel ao contexto ruim, respondeu “5”. O bug não está no modelo nem no prompt — está no **índice desatualizado** (freshness, Cap. 12). Sem o trace, você teria culpado o LLM e ajustado o prompt em vão.

O que é observabilidade agêntica

> **Observabilidade** é a capacidade de explicar o comportamento interno de um sistema a partir do que ele emite — logs, métricas e traces. Para agentes, acrescenta-se uma dimensão: *por que* o modelo decidiu o que decidiu, e *quanto* custou.

Os três pilares clássicos, aplicados a agentes:

- **Tracing** — a sequência de passos de uma execução, como *spans* aninhados (rewrite → retrieval → tool call → geração). É o que reconstrói a "linha de raciocínio".
- **Logging** — o registro detalhado de cada passo: o prompt exato enviado, a resposta crua, os argumentos do tool call, o erro.
- **Metrics** — agregados: latência p95, taxa de erro, tokens por request, custo por tenant, taxa de tool call malsucedido.

A esses, a observabilidade *agêntica* soma três que software tradicional não tem:

- **Prompt tracking** — qual versão do prompt/agent gerou esta resposta (você muda prompts toda semana; precisa correlacionar).
- **Agent telemetry** — decisões do agente: qual ferramenta escolheu, quantas rodadas do loop, quando desistiu.
- **Cost tracking** — tokens de entrada/saída e custo por chamada, por sessão, por tenant. Custo é uma métrica de primeira classe (e o assunto do Cap. 17).

Observabilidade tradicional vs agêntica

A diferença não é cosmética — muda o que você precisa registrar e como interpreta:

Dimensão	Tradicional	Agêntica
Determinismo	mesma entrada → mesma saída	mesma entrada → saídas diferentes
Unidade de falha	exceção, status 500	decisão plausível e errada
O que rastrear	request → response	rewrite → retrieve → rerank → tool → gerar
Custo	desprezível por request	**dominante** — precisa ser métrica
Reprodução	re-rodar com a mesma entrada	precisa de prompt + contexto + seed + versão
”Causa raiz”	linha de código	passo do pipeline (retrieval? prompt? modelo?)

A consequência prática: num sistema tradicional, você loga a borda (entrou, saiu). Num agente, você precisa logar o **miolo** — porque a falha mora lá dentro, num passo intermediário que parecia ok.

Como instrumentar, na prática

O padrão da indústria é **OpenTelemetry** com as convenções semânticas de GenAI — assim seus traces de IA entram nas mesmas ferramentas (Grafana, Datadog, Langfuse) que o resto do stack já usa. A unidade é o *span*: cada passo do agente abre um, anota seus atributos e fecha.

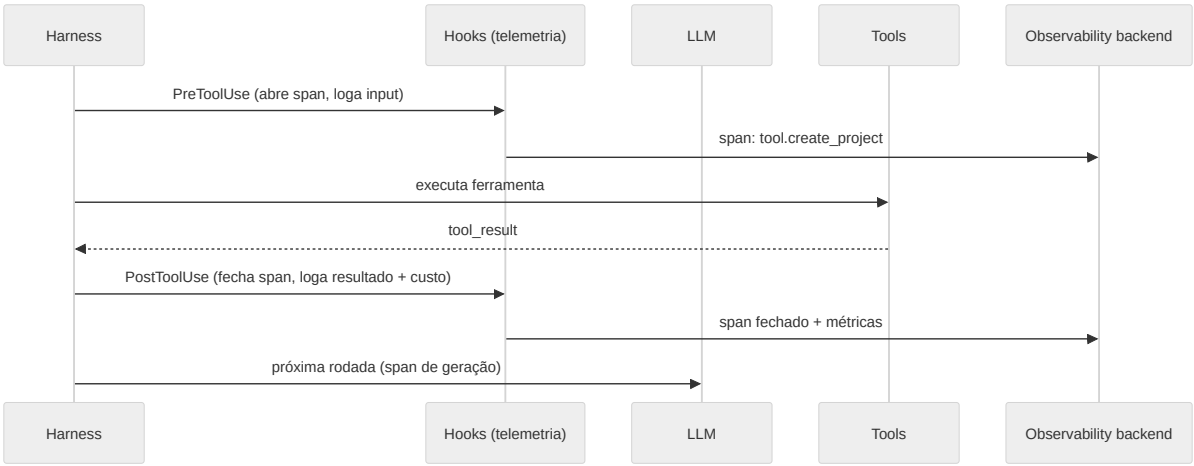
```
// Instrumentar uma chamada ao LLM como um span com atributos de GenAI
async function tracedGenerate(ctx: Ctx, prompt: Prompt) {
  return tracer.startActiveSpan("llm.generate", async (span) => {
    span.setAttributes({
      "gen_ai.system": "anthropic",
      "gen_ai.request.model": ctx.model,         // ex.: claude-sonnet-4-6
      "gen_ai.prompt.version": ctx.promptVersion, // prompt tracking
      "ignition.tenant_id": ctx.tenantId,         // correlação por tenant
    });
    try {
      const res = await llm.generate(prompt);
      span.setAttributes({
        "gen_ai.usage.input_tokens": res.usage.input,
        "gen_ai.usage.output_tokens": res.usage.output,
        "ignition.cost_usd": estimateCost(ctx.model, res.usage), // cost tracking
      });
      return res;
    } catch (err) {
      span.recordException(err);          // logging do erro no trace
      throw err;
    } finally {
      span.end();
    }
  });
}
```

Quatro disciplinas que tornam o trace útil em vez de barulhento:

1. **Correlacione tudo por `trace_id` e `tenant_id`.** Uma execução = um trace; todo span carrega o tenant. É o que permite filtrar “todos os erros do tenant Acme”.
2. **Versione o prompt no span.** Sem `prompt.version`, você não consegue dizer se o bug nasceu na mudança de ontem.
3. **Registre tokens e custo em cada chamada.** Custo invisível é custo descontrolado (Cap. 17).
4. **Nunca logue PII ou segredo cru.** O prompt pode conter dado pessoal (Cap. 13, LGPD) e o tool result pode conter uma chave. Redija antes de persistir.

Conectando ao Harness

Onde a observabilidade se conecta ao agente é o **harness** (Cap. 02) — e especificamente os **hooks** (Cap. 07). O harness é quem intercepta cada passo do loop; logo, é o ponto natural para emitir telemetria sem poluir a lógica do agente.



A beleza do padrão: o agente (o `.md` do Cap. 03) não sabe que está sendo observado. Os hooks `PreToolUse`/`PostToolUse` do harness emitem os spans automaticamente. Observabilidade vira uma propriedade da **plataforma**, não uma responsabilidade que cada agente precisa lembrar de implementar — exatamente como deveria ser num SaaS multi-tenant como a IgnitionStack, onde dezenas de agentes compartilham a mesma instrumentação.

E fecha o ciclo com o capítulo anterior: os sinais de produção que alimentam os evals (Cap. 15) *saem* daqui. O trace que mostrou “retrieval com score 0.48” vira um novo caso de golden dataset. Observabilidade não só explica o passado — abastece a próxima rodada de qualidade.

Trade-offs e armadilhas

- **Logar a borda não basta.** “Entrou X, saiu Y” não diz em qual dos dez passos a coisa desandou. Instrumente o miolo.
- **Trace sem versão de prompt é um trace cego.** Você muda prompts toda semana; sem correlacionar, não sabe o que causou o quê.
- **Custo não medido é custo descontrolado.** Se você não registra tokens por chamada, a primeira vez que vê o número é na fatura.
- **PII no log é incidente de privacidade.** Prompts e resultados carregam dado pessoal e segredo. Redija na origem, antes de persistir.
- **Telemetria demais é seu próprio custo.** Logar cada token de cada request em alto volume tem custo de storage e de processamento. Amostre o que for caro; mantenha 100% só no que importa (erros, custo).
- **Observabilidade não conserta nada — só mostra.** Ela aponta o passo culpado; corrigir ainda é com você.

Como saber se você entendeu

Você dominou este capítulo se consegue:

- ler um trace de agente e localizar em qual span a falha nasceu;
- listar o que a observabilidade agêntica registra além dos três pilares clássicos;
- explicar por que os hooks do harness são o lugar certo para emitir telemetria.

Fontes

- OpenTelemetry — Semantic Conventions for Generative AI (spans, atributos `gen_ai.*`): https://opentelemetry.io/docs/specs/semconv/gen-ai/ (https://opentelemetry.io/docs/specs/semconv/gen-ai/)
- Anthropic — uso da API e tracking de `usage` (input/output tokens): https://docs.anthropic.com/en/api/messages (https://docs.anthropic.com/en/api/messages)

- Langfuse — observabilidade open-source para aplicações LLM (traces, custo, prompt management): https://langfuse.com/docs (https://langfuse.com/docs)
- Honeycomb — “Observability: a Manifesto” (os fundamentos de traces e cardinalidade): https://www.honeycomb.io/blog/observability-a-manifesto (https://www.honeycomb.io/blog/observability-a-manifesto)

Síntese

Observabilidade agêntica torna o agente legível depois do fato: traces reconstroem a sequência de decisões, prompt tracking correlaciona comportamento a versão, cost tracking impede surpresas na fatura. Emitida pelos hooks do harness, vira propriedade da plataforma — todo agente da IgnitionStack instrumentado de graça. Foi o trace que revelou o retrieval fraco, não o palpite. E uma das métricas que esse trace expõe — o custo por chamada, por usuário, por tenant — é tão central que merece um capítulo só dela.

Próximo: Capítulo 17 — Cost Engineering.

Capítulo 17 — Cost Engineering

Como tornar um produto de IA economicamente viável: custo por request, por usuário e por agente, sem sacrificar qualidade.

> Um produto de IA que não fecha a conta de custo não é um produto — é um experimento subsidiado. Cost engineering é o que separa uma feature de IA que escala de uma que quebra a margem no primeiro mês de tração.

TL;DR: Cost engineering é tratar custo de inferência como uma métrica de produto: medir por request/usuário/agente e atacar com cache, batching, model routing e otimização de tokens — sem degradar a qualidade que os evals (Cap. 15) protegem.

Este é o último capítulo, e fecha a Parte II amarrando tudo. Você já sabe fazer o agente recuperar, lembrar, agir, medir e observar. Falta a pergunta que decide se ele *sobrevive* em produção: **isso fecha a conta?** Num SaaS multi-tenant como a IgnitionStack, onde milhares de usuários disparam agentes o dia todo, custo não é detalhe de fim de mês — é uma decisão de arquitetura desde o primeiro span.

Primeiro, o custo em ação

A IgnitionStack lançou um assistente de código. Sucesso de uso, desastre de margem. A observabilidade (Cap. 16) revelou o porquê:

```
ANTES (ingênuo)                          unit economics
──────────────────────────────────────────────────────

toda request → opus, contexto cheio
  custo médio/request:   $0.090
  requests/usuário/mês:  400
  custo/usuário/mês:     $36.00
  preço do plano Pro:    $29.00
  margem:                −$7.00   ✗  perde dinheiro por usuário ativo
```

Cada usuário Pro engajado **dava prejuízo**. A reação errada seria cortar a feature ou subir o preço. A reação de engenharia foi atacar o custo por partes — cache, routing, batching, poda de contexto — sem mexer no preço nem (segundo os evals) na qualidade:

```
DEPOIS (engenheirado)                    unit economics
──────────────────────────────────────────────────────

routing + prompt cache + contexto podado
  custo médio/request:   $0.011    ▼ 88%
  custo/usuário/mês:     $4.40
  margem (plano $29):    +$24.60   ✓  feature lucrativa
```

Mesmo produto, mesma qualidade nos evals, **8x mais barato**. Nada disso foi mágica — foi medir e aplicar quatro alavancas conhecidas.

O que é cost engineering

> **Cost engineering** é a disciplina de medir e otimizar o custo de inferência de um sistema de IA como uma propriedade de produto — em termos de unidade de negócio (request, usuário, agente, tenant), não só de tokens.

O erro mental mais comum é pensar em “preço por milhão de tokens”. O número que importa é o **custo por unidade de negócio**:

Unidade	Pergunta que responde	Por que importa
Custo por request	quanto custa uma interação?	otimização técnica
Custo por usuário/mês	esse usuário dá lucro?	viabilidade do plano
Custo por agente	qual agente é caro demais?	onde otimizar primeiro
Custo por tenant	esse cliente é rentável?	pricing e limites

Sem mapear custo para essas unidades (o que a observabilidade do Cap. 16 torna possível), você “economiza tokens” no lugar errado e ignora o agente que está sangrando a margem.

As quatro alavancas

1. Cache — não pague duas vezes pelo mesmo contexto

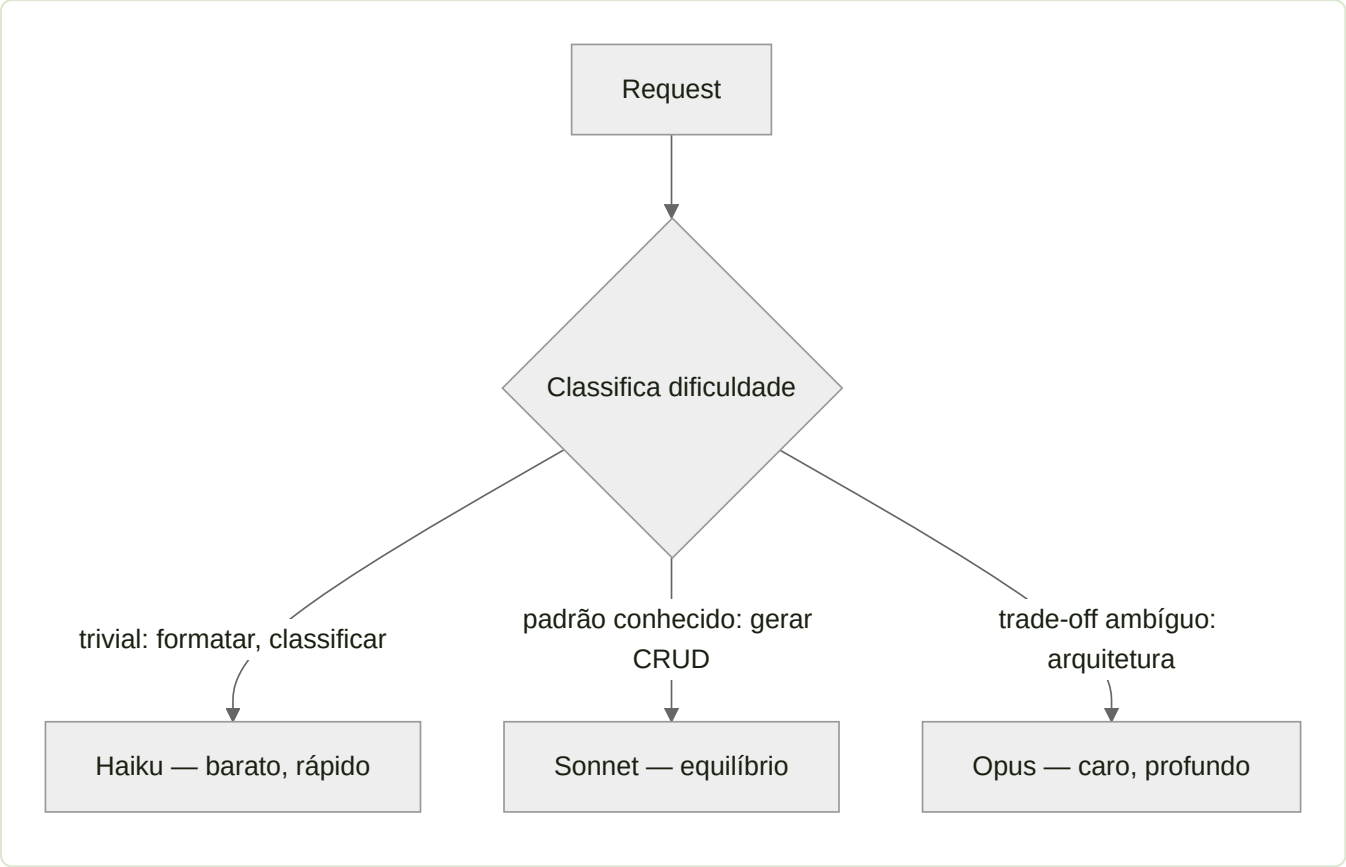
Grande parte do prompt de um agente é **estável**: o system prompt, as definições de ferramentas, os exemplos, os docs do RAG. *Prompt caching* faz o provedor reusar o processamento desse prefixo — leituras de cache custam uma fração do preço normal de entrada.

```
// Marca o prefixo estável como cacheável: system + tools + contexto reusável
const message = {
  system: [
    { type: "text", text: SYSTEM_PROMPT },
    { type: "text", text: TOOL_DEFINITIONS, cache_control: { type: "ephemeral" } },
  ],
  // só a parte variável (a pergunta do usuário) paga preço cheio de entrada
  messages: [{ role: "user", content: userQuestion }],
};
```

Na IgnitionStack, onde milhares de requests compartilham o mesmo system prompt e as mesmas tool definitions, o cache sozinho cortou boa parte do custo de entrada. Regra: **o que se repete entre chamadas deve ser cacheado.**

2. Model routing — o modelo certo para a tarefa certa

Você viu no Capítulo 03 que casar `model` ↔ dificuldade é design. Em custo, é a maior alavanca. Modelos diferem em ordens de magnitude de preço, e usar o topo de linha para tudo é desperdício:



A maioria das requests é mais simples do que parece. Classificar pedidos triviais (“isso é uma pergunta de FAQ?”) com Haiku, deixar geração de CRUD no Sonnet e reservar Opus para decisões de arquitetura é o que derrubou o custo médio no exemplo de abertura — sem que os evals (Cap. 15) acusassem queda de qualidade nas tarefas que importam. **Faça o roteamento provar-se no eval**, não no achismo.

3. Batching — desconto por não ter pressa

Nem toda tarefa é interativa. Reindexar embeddings (Cap. 11), rodar a suíte de evals (Cap. 15), gerar resumos noturnos — nada disso precisa de resposta em tempo real. APIs de *batch* processam esse trabalho assíncrono com desconto expressivo (tipicamente ~50%) em troca de latência maior. **Trabalho que pode esperar não deve pagar preço de tempo real.**

4. Token optimization — menos tokens, mesma resposta

Cada token na janela é pago e processado (Cap. 05). Reduzir tokens sem perder sinal é economia pura:

- **Poda de contexto.** Não despeje 20 chunks quando 5 (pós-rerank, Cap. 12) bastam. O reranking é também uma alavanca de custo.
- **Compactação.** Resuma histórico antigo da conversa em vez de carregá-lo inteiro (Cap. 05).
- **Saída enxuta.** Tokens de *saída* costumam custar mais que os de entrada. Peça respostas concisas; use structured output (Cap. 14) em vez de prosa quando a saída é consumida por máquina.
- **Proxies de compressão.** Ferramentas que comprimem outputs verbosos de terminal/logs antes de mandar ao modelo (o padrão RTK citado no Cap. 10) cortam tokens em tarefas de desenvolvimento.

Comparando provedores

A escolha de provedor é parte do cost engineering — mas o eixo certo é **custo × qualidade × latência**, não preço isolado. Um modelo "barato" que erra e exige retry sai caro; um "caro" que resolve de primeira pode ser o mais econômico por tarefa resolvida.

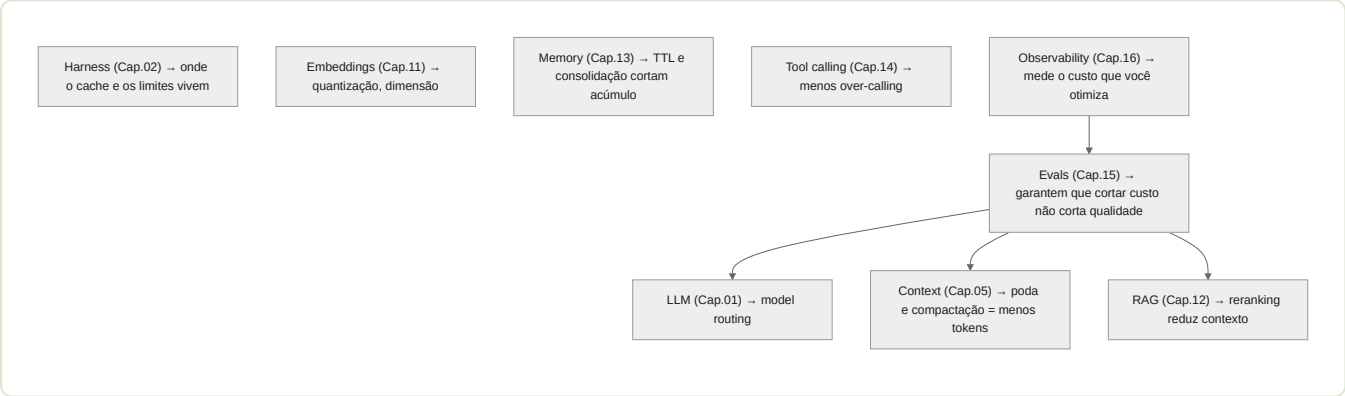
Família	Perfil de custo	Força	Quando faz sentido
Claude (Opus/Sonnet/Haiku)	premium no topo, Haiku competitivo	raciocínio, código, tool use, prompt caching	agentes de produção, código
GPT	amplo leque de tiers	ecossistema, structured outputs	uso geral, integração ampla
Gemini	agressivo em contexto longo	janelas enormes, multimodal	grandes volumes de contexto
Modelos locais (Llama, etc.)	sem custo por token; custo de infra	privacidade, dado nunca sai	alto volume previsível, dado sensível (LGPD, Cap. 13)

> Preços absolutos mudam toda hora — por isso este livro evita cravá-los. O que **não** muda é o método: meça custo por unidade de negócio, e escolha o modelo que minimiza custo *por tarefa resolvida no eval*, não por milhão de tokens na tabela.

Modelos locais merecem nota: eliminam custo por token e mantêm o dado dentro de casa (resolvendo de uma vez parte da LGPD do Cap. 13), mas trocam isso por custo de GPU, ops e, em geral, qualidade menor. Faz sentido em volume alto e previsível, ou quando o dado não pode sair — raramente como primeira escolha de um time pequeno.

Conectando ao stack inteiro

Cost engineering não é uma camada — é uma lente sobre todas as nove. Cada capítulo deste livro tem uma alavanca de custo embutida:



Note a dupla central: **observabilidade mede** o custo (Cap. 16) e **evals garantem** que a otimização não degrada qualidade (Cap. 15). Sem esses dois, "cost engineering" vira corte cego — você economiza e quebra o produto sem perceber. Com eles, vira o que foi no exemplo de abertura: 88% mais barato, mesma qualidade medida. **Custo é uma feature de arquitetura, e como toda feature, precisa de teste e de telemetria.**

Trade-offs e armadilhas

- **Otimizar custo sem eval é cortar qualidade às cegas.** Toda economia precisa passar pela suíte do Cap. 15. Barato e errado é o pior dos mundos.
- **O modelo mais barato nem sempre é o mais econômico.** Se erra e exige retry ou intervenção humana, o custo total por tarefa resolvida sobe. Meça por resultado, não por token.
- **Cache mal desenhado não economiza.** Se a parte cacheável muda a cada request, não há reuso. Estruture o prompt com o estável no prefixo.
- **Custo invisível é custo descontrolado.** Sem o tracking do Cap. 16, você descobre o problema na fatura. Instrumente antes de escalar.

- **Premature optimization também vale aqui.** Não roteie nem cacheie no MVP de 10 usuários. Otimize quando a observabilidade mostrar onde dói — primeiro meça, depois corte.
- **Pricing precisa caber no custo.** Se o plano não cobre o custo por usuário, nenhuma otimização salva para sempre. Cost engineering informa o pricing, não só a infra.

Como saber se você entendeu

Você dominou este capítulo se consegue:

- calcular o custo por usuário/mês e dizer se um plano é rentável;
- escolher entre cache, routing, batching e poda de tokens para um gargalo dado;
- explicar por que toda otimização de custo precisa passar pelos evals.

Fontes

- Anthropic — Prompt caching (como cachear prefixos estáveis e o impacto no custo): https://docs.anthropic.com/en/docs/build-with-claude/prompt-caching (https://docs.anthropic.com/en/docs/build-with-claude/prompt-caching)
- Anthropic — Message Batches API (processamento assíncrono com desconto): https://docs.anthropic.com/en/docs/build-with-claude/batch-processing (https://docs.anthropic.com/en/docs/build-with-claude/batch-processing)
- Anthropic — Models overview (tiers Opus/Sonnet/Haiku e trade-offs de capacidade): https://docs.anthropic.com/en/docs/about-claude/models (https://docs.anthropic.com/en/docs/about-claude/models)
- Anthropic — “Building effective agents” (simplicidade e parcimônia como princípio de custo): https://www.anthropic.com/research/building-effective-agents (https://www.anthropic.com/research/building-effective-agents)

Síntese

Cost engineering fecha a Parte II porque fecha a conta: medir custo por unidade de negócio e atacá-lo com cache, model routing, batching e otimização de tokens é o que torna a IA da IgnitionStack lucrativa em vez de subsidiada — 88% mais barata, mesma qualidade nos evals. Custo é uma feature de arquitetura, medida pela observabilidade (Cap. 16) e protegida pelos evals (Cap. 15).

Com isso, o stack está completo nas duas partes: a **Parte I** montou o sistema agêntico — do LLM ao CLI, com o agent no centro. A **Parte II** colocou esse sistema em produção — recuperando conhecimento (embeddings, RAG), lembrando do relacionamento (memory), agindo com segurança (tool calling), e provando-se confiável, observável e econômico (evals, observability, cost). Um agente é um arquivo; um *produto* de IA é esse arquivo cercado pela disciplina destes sete capítulos.

Voltar ao índice.