

Engenharia de agentes de IA

Projetar, construir, avaliar e operar
agentes que você pode verificar



Enrique Gutiérrez

Engenharia de agentes de IA

Projetar, construir, avaliar
e operar agentes que você pode verificar

Enrique Gutiérrez

Engenharia de agentes de IA

Projetar, construir, avaliar e operar agentes que você pode verificar

Tradução de *AI Agents, Engineered: Designing, Building, Evaluating, and Operating Agents You Can Verify* (primeira edição, 2026). Tradução do autor.

Copyright © 2026 Enrique Gutiérrez. Todos os direitos reservados.

Nenhuma parte deste livro pode ser reproduzida ou transmitida de qualquer forma sem a autorização prévia e por escrito do autor, exceto citações breves em resenhas e trabalhos acadêmicos. O material citado continua sendo propriedade de seus respectivos autores e é referenciado nas notas.

Os nomes de produtos e empresas mencionados neste livro são usados apenas como exemplos e podem ser marcas registradas de seus proprietários. O autor empenhou-se ao máximo para garantir a exatidão das informações aqui contidas; no entanto, elas são fornecidas sem garantia, e o autor não se responsabiliza por quaisquer danos decorrentes de seu uso.

Publicado no Leanpub. Primeira edição, 2026.

Sumário

Prefácio	ix
I Fundamentos	1
1 O que é um agente?	2
1.1 A ideia de agente	2
1.2 Agentes, workflows e chatbots	5
1.3 Você precisa mesmo de um agente?	8
2 O motor: como funcionam os modelos de linguagem	12
2.1 LLMs como preditores do próximo token	12
2.2 Os tokens e a janela de contexto	16
2.3 Como o texto é gerado	20
2.4 Fundamentos da escrita de prompts	24
2.5 Saída estruturada e chamada de função	28
2.6 Limitações e modos de falha	31
II Construção de agentes	37
3 O loop do agente	38
3.1 O loop	38
3.2 ReAct: intercalar raciocínio e ação	42
3.3 Um agente mínimo do zero	46
3.4 Condições de parada, orçamentos e frameworks	50
4 Planejamento, raciocínio e autocorreção	54
4.1 Decomposição de tarefas e planejamento	54
4.2 Técnicas de raciocínio	61
4.3 Reflexão e autocorreção	65

5	As ferramentas e o espaço de ações	72
5.1	Escolher o conjunto e a granularidade das ferramentas	73
5.2	A interface da ferramenta	78
5.3	Adaptar programas feitos para humanos	81
5.4	A execução de código como ação universal	84
5.5	Uso do computador e automação de navegador	87
6	Habilidades, protocolos e interoperabilidade	93
6.1	O que é uma habilidade	94
6.2	Divulgação progressiva	98
6.3	Habilidades versus ferramentas versus protocolos	102
6.4	Protocolos e interoperabilidade	105
6.5	Padrões de projeto de habilidades e composição	109
III	Engenharia de contexto	115
7	Gerenciamento da janela de contexto	116
7.1	Engenharia de contexto versus engenharia de prompt	117
7.2	Um recurso escasso e limitado pela atenção	120
7.3	Como o contexto falha	123
7.4	As quatro operações: escrever, selecionar, comprimir, isolar	129
8	Recuperação e conhecimento	134
8.1	Geração aumentada por recuperação	135
8.2	Além da recuperação ingênua	139
8.3	Recuperação agêntica	143
8.4	RAG versus contexto longo versus fine-tuning	146
9	Memória: o estado de trabalho em execuções longas	150
9.1	A taxonomia da memória	151
9.2	Arquiteturas de memória	154
9.3	Manter saudável uma execução longa	159
9.4	Isolamento e quarentena	163
IV	Padrões e escolhas	168
10	Workflows e padrões de composição	169
10.1	O LLM aumentado como unidade básica	170
10.2	Encadeamento de prompts, roteamento e paralelização	172
10.3	O loop avaliador-otimizador	180

11 Sistemas multiagente	186
11.1 O padrão orquestrador-worker	187
11.2 Um contexto focado e autocontido para cada worker	191
11.3 Modos de falha da coordenação	195
11.4 Arquétipos de subagente com nome	198
11.5 Agentes em paralelo orquestrados por um humano	200
11.6 O muro do merge e o veredito: um agente ou vários	204
12 Supervisão e autonomia	209
12.1 Pontos de aprovação e escalonamento	211
12.2 Revisar a intenção e os resultados, não as linhas	215
12.3 Interrupção, redirecionamento e o dial de autonomia	220
13 Escrever o loop externo	225
13.1 Uma breve história do loop	226
13.2 A função objetivo	229
13.3 Engenharia de loops	232
13.4 Quando os loops se compõem	236
13.5 Manter o loop seguro	240
14 Escolher a abordagem: quando <i>não</i> construir um agente	245
14.1 A escada revisitada	246
14.2 Indícios de que você precisa de um, indícios de que não	249
V Agentes confiáveis	254
15 Observabilidade e depuração	255
15.1 Traces e spans	256
15.2 Monitoramento em produção	262
15.3 Reproduzir falhas não determinísticas	266
15.4 Uma taxonomia dos bugs comuns de agentes	271
16 Avaliação de agentes	277
16.1 Por que avaliar é difícil	278
16.2 Construir um conjunto de avaliação	282
16.3 O LLM como juiz	285
16.4 Desenvolvimento orientado a avaliações e barreiras de regressão	290
17 Segurança, proteção e guardrails	294
17.1 Injeção de prompt	295
17.2 A tríade letal	299
17.3 Guardrails como camada separada e determinística	303
17.4 Privilégio mínimo, sandbox e aprovação humana	307
17.5 A cadeia de suprimentos	311

18 Confiabilidade, estado e o harness	317
18.1 Devolver os erros ao modelo	318
18.2 Retentativas, timeouts e orçamentos	321
18.3 Estado e recuperação de falhas	323
18.4 Fallbacks, degradação e falha parcial	327
18.5 Engenharia de harness	329
18.6 Quanto harness construir	333
 VI Produção e operação	 337
19 Custo, latência e desempenho	338
19.1 Para onde vão os tokens (e o dinheiro)	339
19.2 Alavancas de custo	343
19.3 Latência	347
19.4 O triângulo acurácia-custo-latência	351
20 Implantação e escalabilidade	355
20.1 Formatos de atendimento	356
20.2 Execução durável e estado em escala	360
20.3 Execução em sandbox em escala	363
20.4 Limites de taxa, filas e contrapressão	365
20.5 A escada de implantação	367
 VII Aplicações e o caminho à frente	 372
21 Agentes de programação	373
21.1 Por que a programação saiu na frente	374
21.2 O loop do agente de programação e o leque de tarefas	377
21.3 Programação aumentada versus vibe coding	380
21.4 Tornar o código legível para agentes	385
21.5 A lição geral	389
22 O workflow de programação na prática	392
22.1 O loop canônico: brainstorm, especificação, plano, execução	393
22.2 Desenvolvimento orientado a especificação	398
22.3 Desenvolvimento orientado a testes com agentes	401
22.4 O Git como rede de segurança	404
22.5 QA e revisão agênticos	408
22.6 Entender código que você não escreveu	410

23 Agentes de pesquisa e de negócios	416
23.1 Agentes de pesquisa e de informação	417
23.2 A lacuna de verificação	420
23.3 Agentes de negócios e operacionais	425
23.4 Pesar o retorno sobre o investimento contra o risco	430
24 UX de agentes e confiança humana	434
24.1 Calibração da confiança	435
24.2 Transparência e feedback de progresso	439
24.3 Redirecionamento e falha graciosa	442
24.4 De fazer a dirigir	446
25 Receitas transversais: grandes padrões que atravessam setores	450
25.1 O controle deslizante de autonomia	451
25.2 Autoresearch	453
25.3 A base de conhecimento pessoal que se mantém sozinha	456
25.4 O analista de pesquisa aprofundada	459
25.5 O vigia em segundo plano	460
25.6 Do não estruturado ao estruturado, em escala	462
25.7 A triagem e o roteador de filas	464
25.8 O colega de trabalho digital	465
25.9 O premortem e o simulador de red team	467
26 Agentes como classificadores e pontuadores	472
26.1 O classificador sem conjunto de treinamento	473
26.2 Exemplos como livro didático, não como banco de dados	476
26.3 Primeiro raciocinar, depois explicar	478
26.4 Dos rótulos às pontuações	480
26.5 Desenvolvê-lo sem se enganar	482
26.6 Empacotá-lo, e quando recorrer a ele	485
27 A fronteira e como continuar aprendendo	489
27.1 Combinar a generatividade com verificadores determinísticos	490
27.2 O ecossistema como camadas duráveis	496
27.3 Construir versus adotar	500
27.4 Tendências duráveis, e como continuar aprendendo	503
VIII Apêndices	509
A Um agente mínimo, comentado	510
A.1 O programa completo	511
A.2 O cliente do modelo	513
A.3 O registro de ferramentas	513

A.4	Executar uma solicitação	515
A.5	O loop	516
A.6	As instruções permanentes	517
A.7	Uma execução, de ponta a ponta	518
A.8	O que falta de propósito	520
B	Glossário	521
C	Leituras complementares comentadas e mapa de fontes	536
C.1	Como a biblioteca está organizada	537
C.2	O mapa capítulo a capítulo	537
C.3	Referências primárias duráveis	541
C.4	Uma segunda prateleira: fundamentos pressupostos e debates em andamento	544

Prefácio

Este livro existe por causa de um momento específico, e suspeito que você já passou por ele. Você entrega a um modelo de linguagem uma tarefa que tomaria meia hora de uma pessoa: descobrir por que o build quebrou, digamos, ou conciliar uma planilha bagunçada com o banco de dados. Em vez de uma resposta, você o vê *trabalhar*. Ele pesquisa, lê, tenta alguma coisa, esbarra em um erro, se corrige e volta alguns minutos depois, com tudo pronto. A primeira sensação é de encanto. A segunda chega um instante depois e dura mais: *Eu não vi todos os passos. Até que ponto posso confiar nisso?*

Essa segunda sensação é o assunto deste livro.

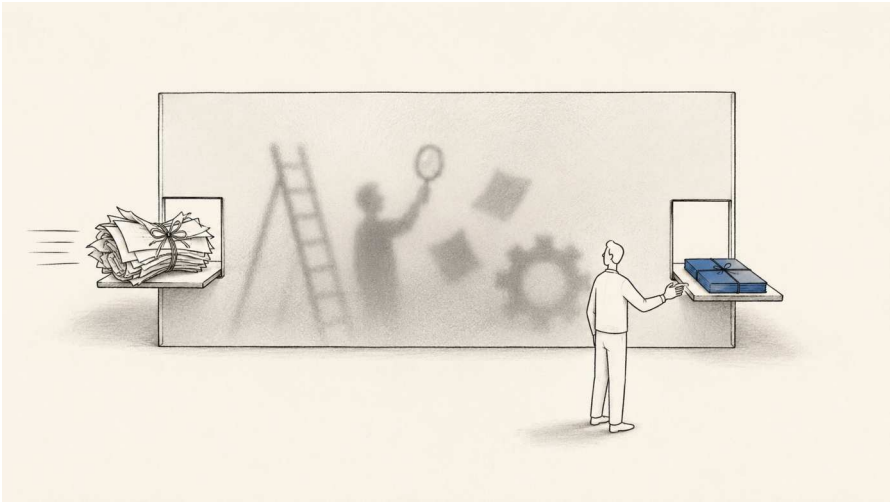


Figura 1: O trabalho bagunçado entra por um lado de uma parede de vidro fosco e o trabalho pronto sai pelo outro; os passos no meio são apenas silhuetas. O encanto é imediato. A dúvida dura mais que ele, e é o assunto deste livro.

Escrevi este livro para quem trabalha com engenharia no dia a dia — alguém à vontade com software, mas não necessariamente com aprendizado de máquina — que está avaliando agentes, ou acabou de começar a construir

com eles, e quer deixar para trás tanto o hype quanto o pavor e chegar à engenharia que está por baixo. Você vai encontrar definições construídas a partir dos primeiros princípios, padrões que se repetem em todos os domínios em que agentes fazem trabalho útil, exemplos em pseudocódigo simples e um interesse persistente em quanto as coisas custam e em como elas falham. O único pré-requisito é a bagagem normal de quem trabalha com engenharia; sempre que aparece um termo técnico, eu o defino na hora, em palavras simples.

Duas promessas moldam todos os capítulos, e quero deixá-las explícitas, porque juntas elas explicam o que você vai e o que não vai encontrar aqui.

A primeira promessa é que o livro é *agnóstico quanto ao tempo*. Esta área avança em um ritmo que constrange qualquer um que ponha as coisas no papel; enquanto eu reunia material para estes capítulos, anotações que eu tinha feito poucos meses antes já estavam ficando velhas, com produtos rebatizados e números de benchmark superados. Por isso me recusei a ancorar qualquer coisa na época em que foi escrita. Nenhuma afirmação deste livro depende de qual modelo é o melhor no momento, de quanto custa um token neste trimestre ou do tamanho das janelas de contexto deste ano. Onde aparece um número, ele está marcado como ilustração. O objetivo é um livro que se leia tão bem daqui a alguns anos quanto hoje.

A segunda promessa é que ele é *agnóstico quanto à solução*. O conhecimento durável desta área está em conceitos e padrões — o loop, o contrato da ferramenta, o orçamento de contexto, a suíte de avaliação — e esses valem para qualquer fornecedor e qualquer framework. A sintaxe de um produto específico não se transfere, e envelhece em meses. Por isso este livro ensina o padrão e cita ferramentas concretas só de passagem, como exemplos, assim identificados, de uma categoria. Ao terminar, você deve ser capaz de se sentar diante de qualquer *harness* (a casca de software dentro da qual um agente roda) que a sua equipe tenha adotado, ou diante de nenhum, e saber o que construir.

Uma ideia volta tantas vezes nestas páginas que equivale a uma tese, e é bom que você a conheça desde já: *um agente só é tão confiável quanto o sinal que você tem para verificá-lo*. Um modelo de linguagem produz texto fluente e seguro de si, esteja certo ou errado; o tom dele não diz nada em que você possa se apoiar. A confiança que você deposita em um agente, portanto, tem de vir de algum lugar fora do modelo: uma suíte de testes que passa, um schema que valida, uma fonte que você pode abrir e conferir, um humano que aprova o passo irreversível. Onde existe um sinal desses, você pode conceder autonomia com generosidade — e é em boa parte por isso, como veremos, que a programação se tornou o primeiro domínio em que os agentes provaram seu valor, já que o código chega com o próprio veredito: compile, rode os testes. Onde não existe sinal, a autonomia é um salto de fé, e este livro vai continuar dizendo isso. Quase toda questão de projeto nos capítulos adiante se resolve com o mesmo primeiro movimento: perguntar que sinal indica que deu certo.

Uma palavra sobre vocabulário. “Agente” talvez seja o termo mais sobrecarregado do software moderno; ele é aplicado com a mesma segurança a um bot de roteiro fixo que percorre um menu de atendimento telefônico e a um sistema em que se confia para trabalhar sozinho por uma tarde inteira. Jargão só é útil quando quem fala e quem ouve compartilham uma definição; do contrário, duas pessoas podem discutir apaixonadamente coisas completamente diferentes.¹ Por isso o primeiro capítulo faz o trabalho pouco glamoroso de fixar os termos, e o resto do livro se mantém fiel a eles.

Sobre como ler o livro: ele está dividido em sete partes, ordenadas como um curso. A Parte I estabelece o que é um agente e como o motor que está por baixo — o modelo de linguagem — realmente funciona. As Partes II e III constroem um: o loop, as ferramentas, as habilidades (*skills*) reutilizáveis e a gestão cuidadosa da memória de trabalho do modelo. A Parte IV trata de arquitetura: workflows, sistemas multiagente, o dial entre supervisão e autonomia, o ofício de escrever o loop externo do harness e a disciplina de perguntar se a coisa mais simples que funciona precisa mesmo de um agente. As Partes V e VI cobrem aquilo que acaba sendo a maior parte da engenharia de agentes na vida real: avaliação, observabilidade, segurança, confiabilidade, custo e implantação. A Parte VII se volta para as aplicações e para o que parece durável no caminho à frente. Os apêndices trazem um agente mínimo em pseudocódigo comentado, um glossário e um mapa de leituras complementares. Leia as partes em ordem para ter uma base ou, depois de se orientar, consulte-as como obra de referência; os capítulos remetem uns aos outros e procuram se sustentar sozinhos.

Preciso ser honesto sobre o que um livro como este pode prometer. Alguns detalhes daqui serão superados, provavelmente mais cedo do que eu gostaria; procurei escrever apenas as partes que sobrevivem aos próprios exemplos e sinalizar minha incerteza onde ela existe. A aposta deste livro é que princípios duram mais que produtos. Se, daqui a alguns anos, os exemplos parecerem antiquados, mas as perguntas — *que sinal verifica isto? qual é a coisa mais simples que funciona?* — continuarem sendo as que você faz, o livro terá cumprido seu papel.

¹Simon Willison, “I think ‘agent’ may finally have a widely enough agreed upon definition to be useful jargon now”, simonwillison.net (setembro de 2025). A observação sobre o jargão compartilhado e a definição prática que este livro adota no capítulo 1 são dele.

Parte I

Fundamentos

Capítulo 1

O que é um agente?

Este capítulo cumpre uma promessa feita no prefácio: antes de construir qualquer coisa, definimos com precisão o que um agente *é*. A necessidade é prática e imediata. Fornecedores chamam de “agentes” bots de roteiro fixo; pesquisadores reservam a palavra para sistemas que rodam por horas sem ninguém acompanhando; e, em algum ponto entre os dois, você precisa tomar decisões de verdade com orçamentos de verdade. Se o termo continuar nebuloso, você vai recorrer a um agente autônomo onde um pipeline de três passos teria sido mais barato e mais fácil de depurar, ou vai montar à mão um pipeline rígido para um problema cujos passos você realmente não tem como prever. O capítulo faz três coisas. Apresenta a ideia de agente em si, explica por que ela se tornou viável quando se tornou — sem datas, de propósito — e expõe a bússola pela qual o livro vai se orientar. Em seguida, traça a distinção prática entre chatbots, workflows e agentes, na qual todos os capítulos seguintes se apoiam. E termina com a pergunta que pouca gente faz em voz alta: se você precisa mesmo de um agente.

1.1 A ideia de agente

Deixe de lado as demos e o marketing, e a ideia é pequena o bastante para caber na palma da mão. Um *agente* é um modelo de linguagem colocado em um loop. Ele recebe um objetivo e um conjunto de *ferramentas* — funções que ele pode pedir ao seu programa que execute: pesquise isto, leia aquilo, execute este comando. Ele olha o estado da tarefa, escolhe uma ação, vê o resultado e repete, até julgar que o objetivo foi atingido ou até que uma regra de parada encerre a execução. Depois de anos sem conseguir se entender, a comunidade de quem trabalha na área convergiu em grande parte, no momento em que escrevo, para uma versão de uma frase só: um agente *executa ferramentas*

*em loop para atingir um objetivo.*¹ Cada palavra dessa frase carrega peso. “Ferramentas” é o que conecta o modelo ao mundo; “loop” é o que permite que ele reaja ao que encontra; “um objetivo” é o que encerra o loop — um agente tem uma condição de parada, e é isso que o separa de um processo descontrolado. O capítulo 3 destrincha o loop em código.

O que há de realmente novo aqui é onde mora o fluxo de controle. Em todo programa que você escreveu até hoje, a sequência de operações foi decidida por você, de antemão, em código. Em um agente, a sequência é decidida pelo modelo, em tempo de execução, em resposta ao que ele observa. Essa única mudança é o que faz os agentes merecerem um livro, e também o que os torna caros, lentos e, de vez em quando, alarmantes. O resto do livro trata das duas metades dessa frase.

Você pode, com razão, perguntar por que isso se tornou viável quando se tornou. Os modelos de linguagem já eram fluentes muito antes de alguém confiar um loop a um deles; a fluência nunca foi a peça que faltava. Três coisas precisaram amadurecer. Cada uma vai voltar mais adiante no livro, primeiro como mecanismo e depois como modo de falha, quando começa a faltar.

A primeira: os modelos aprenderam a receber instruções. Um modelo recém-saído do treinamento inicial é um puro continuador de texto — faça uma pergunta e ele pode responder com mais perguntas, porque essa é uma forma plausível de o texto continuar (o capítulo 2 mostra por quê). Uma etapa posterior do treinamento, que a área chama de pós-treinamento (*post-training*), transforma esse continuador em algo a que você pode confiar um trabalho: ele segue orientações, trabalha em direção ao objetivo declarado e para quando termina. Um agente é uma delegação, e delegar exige um trabalhador que aceite a tarefa.

A segunda: o texto ganhou uma ponte para a ação. A *chamada de função* (*function calling*) — o modelo emite uma solicitação legível por máquina, “chame esta função com estes argumentos”, que o seu código executa antes de devolver o resultado a ele — deu ao modelo um canal disciplinado para o mundo. O modelo propõe; o seu código dispõe. O capítulo 2 cobre o mecanismo em detalhe. Aqui, o que importa é a consequência: um sistema que só conseguia escrever *sobre* o mundo passou a poder pedir, em uma forma que um programa consegue obedecer com segurança, que coisas *aconteçam* nele.

A terceira: a memória de trabalho do modelo cresceu. Tudo o que um agente sabe em um dado momento precisa caber na *janela de contexto* do modelo (o trecho limitado de texto que um modelo consegue considerar em uma

¹Simon Willison, “I think ‘agent’ may finally have a widely enough agreed upon definition to be useful jargon now”, simonwillison.net (setembro de 2025). A definição citada — “An LLM agent runs tools in a loop to achieve a goal” (um agente de LLM executa ferramentas em loop para atingir um objetivo) — é dele, defendida ali expressão por expressão. *LLM* é a abreviação usada na área para *large language model* (grande modelo de linguagem), que este livro chama quase sempre de modelo de linguagem, ou simplesmente de modelo.

única chamada). Quando as janelas comportavam umas poucas páginas, um loop morria de fome: as instruções, o histórico e o resultado mais recente de uma ferramenta não cabiam juntos. As janelas cresceram até comportar algo mais próximo de uma sessão de trabalho, e os loops se tornaram sustentáveis.

Cada um desses ingredientes amadureceu como uma curva que cruza um limiar sem fazer barulho, o que é conveniente para nós dois: a explicação do “por que agora” não tem prazo de validade. Ela também diz o que verificar quando alguma nova capacidade é anunciada: pergunte qual dos três ingredientes acabou de melhorar.

Antes da taxonomia e do maquinário, quero entregar a você a bússola pela qual este livro se orienta: quatro rumos que vão reaparecer em quase todos os capítulos. Eu os enuncio aqui sem a defesa completa; a defesa é o livro.

A *verificabilidade* vem primeiro. O prefácio enunciou a tese, e vale repeti-la: um agente só é tão confiável quanto o sinal que você tem para verificá-lo. A convicção com que o próprio modelo fala não traz nenhuma informação sobre se ele está certo, então a confiança precisa vir de fora — um teste que passa, um schema que valida, uma fonte que você pode abrir, um humano que aprova. A primeira pergunta a fazer sobre qualquer projeto de agente é a que este livro vai fazer repetidas vezes: *que sinal indica que deu certo?*

Em seguida vem a *escassez de contexto*. A janela de contexto é finita, e a parte que surpreende as pessoas é que a atenção do modelo se degrada antes de o espaço acabar: um fato enterrado no meio de uma pilha longa fica, na prática, meio esquecido. Trate o contexto como um orçamento a ser gasto com critério. A Parte III deste livro é sobre gastá-lo bem.

O *erro composto* é o terceiro. Um agente encadeia muitos passos, e a confiabilidade se multiplica ao longo deles. Suponha, apenas como ilustração, que cada passo dê certo 95% das vezes; uma execução de vinte passos encadeados dá certo, então, cerca de $0,95^{20} \approx 36\%$ das vezes. Os números específicos variam; a aritmética não, e ela é impiedosa. Loops longos só sobrevivem com recuperação, checkpoints e verificação ao longo do caminho. Este livro dedica capítulos inteiros a esse maquinário.

E o último rumo é a *coisa mais simples que funciona*. O conselho de engenharia que melhor envelheceu nesta área tão jovem é direto: encontre a solução mais simples possível e só acrescente complexidade quando ela comprovadamente compensar — o que pode significar não construir agente nenhum.² Às vezes a coisa mais simples é um único prompt. Às vezes é uma expressão regular. A última seção deste capítulo leva esse rumo a sério o bastante para lhe dar uma discussão própria. A figura 1.1 dispõe os quatro rumos em uma única rosa dos ventos, com a agulha pousada no primeiro

²Erik Schluntz e Barry Zhang, “Building Effective Agents”, blog de engenharia da Anthropic (2024): “recomendamos encontrar a solução mais simples possível e só aumentar a complexidade quando necessário. Isso pode significar não construir sistemas agênticos de forma alguma”. A distinção entre workflow e agente feita no ensaio também ancora a próxima seção.

deles.

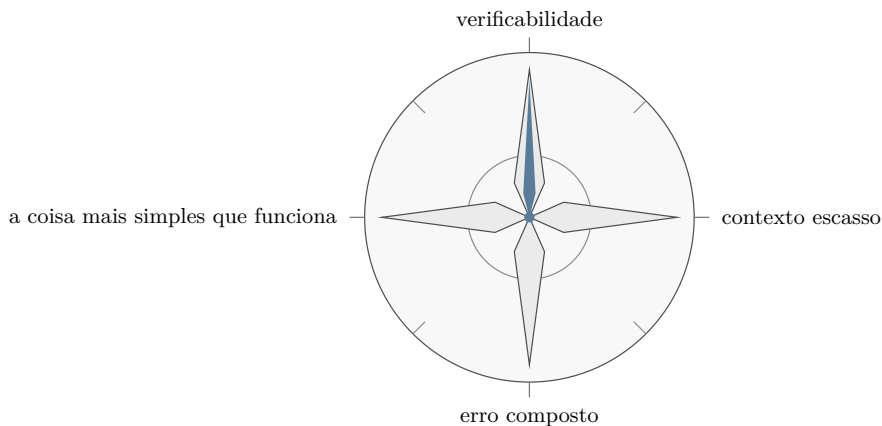


Figura 1.1: A bússola do livro. Quatro rumos decidem as bifurcações de projeto em quase todos os capítulos seguintes, e a agulha repousa sobre o primeiro deles — a verificabilidade — porque um agente só é tão confiável quanto o sinal que você tem para conferi-lo. Os outros três, a escassez de contexto, o erro composto e a coisa mais simples que funciona, são lidos junto com ele, não no lugar dele.

Uma bússola é um instrumento modesto; ela não vai projetar o seu sistema por você. Mas sempre que o livro chegar a uma bifurcação — mais autonomia ou menos, outro agente ou um script simples, contexto mais rico ou mais enxuto — você verá os mesmos quatro rumos decidirem a questão, e espero que, no fim, eles estejam decidindo as suas bifurcações também.

1.2 Agentes, workflows e chatbots

Vou colocar três sistemas na sua frente. Decida qual deles é o agente antes de continuar a leitura.

O primeiro é um assistente de perguntas e respostas. Você cola uma função e pergunta o que ela faz, e ele explica; você cola um diff e pede um rascunho de mensagem de commit, e ele escreve; depois fica esperando a sua próxima mensagem. O segundo processa do mesmo jeito todo ticket de suporte que chega: uma chamada de modelo classifica o ticket, código comum busca o documento de política correspondente, uma segunda chamada de modelo redige uma resposta, uma terceira checa o rascunho com base em uma rubrica, e o resultado cai em uma fila de revisão humana — quatro passos, nessa ordem, todas as vezes. O terceiro recebe a instrução “os testes do checkout estão falhando — corrija” e, sem ninguém acompanhando, pesquisa

na base de código, lê três arquivos, edita um, roda os testes, lê a nova falha, edita de novo e para quando a suíte fica verde.

O instinto da maioria das pessoas separa esses três na hora: o terceiro parece de outra natureza, e o instinto está certo. O trabalho útil é dizer com precisão *por quê*. Os três podem ser construídos exatamente sobre o mesmo modelo. O que muda é a pergunta que a seção anterior apontou como o que há de realmente novo: quem decide o que acontece em seguida. No primeiro sistema, é você, uma mensagem de cada vez. No segundo, é o seu código; o modelo só preenche o conteúdo ao longo de uma rota que foi fixada antes de o primeiro ticket chegar. No terceiro, quem decide é o *modelo*: que arquivo ler, se edita ou roda de novo, quando parar. Essa pergunta — quem é dono do fluxo de controle — é a distinção inteira, e ela nos dá os três termos que este livro usa em toda parte. A figura 1.2 desenha os três lado a lado, diferindo apenas em quem decide o próximo passo.

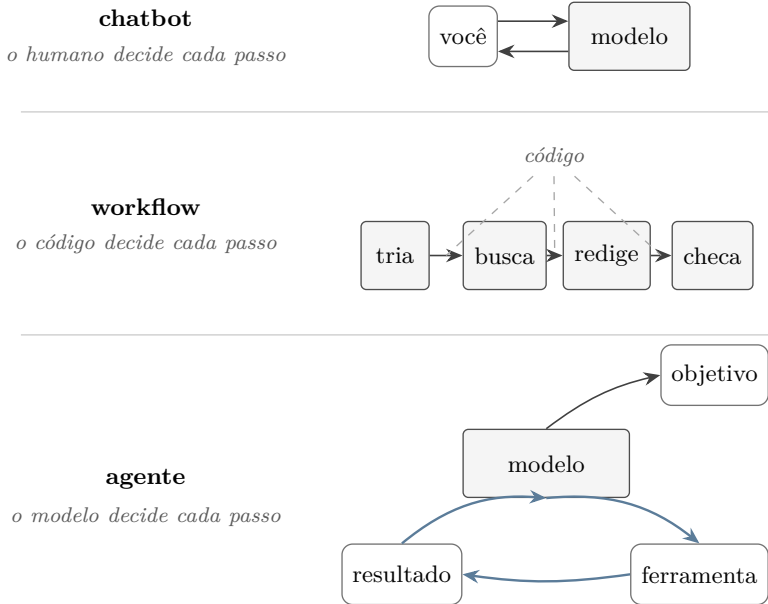


Figura 1.2: Chatbot, workflow e agente diferem em uma única coisa: quem decide o próximo passo. Em um chatbot, o humano conduz cada turno; em um workflow, o caminho está fixado em código e o modelo apenas preenche cada passo; em um agente, o modelo é dono do loop, escolhendo ferramentas e lendo resultados até atingir o objetivo. O loop que pertence ao modelo (na cor de destaque) é o assunto inteiro deste livro.

Um *chatbot* é uma conversa turno a turno. O humano conduz cada passo; o modelo gera texto e espera. Não há loop e nada é executado. Chatbots são

úteis de verdade e muitas vezes a ferramenta certa, e sua superfície de falha é pequena de um jeito que quero deixar registrado: o pior que um chatbot pode produzir é uma resposta errada, porque agir sobre o mundo está fora do alcance dele.

Um *workflow* (fluxo de trabalho) é formado por várias chamadas de modelo (e chamadas de ferramenta) ligadas entre si ao longo de caminhos que o seu código define de antemão. O modelo preenche o conteúdo em cada passo — classifique isto, redija aquilo — mas é o seu código que decide qual passo vem a seguir. Os formatos recorrentes têm nomes que você vai reencontrar no capítulo 10: *encadeamento* (a saída de uma chamada alimenta a seguinte), *roteamento* (classificar e então ramificar) e *paralelização* (disparar várias chamadas ao mesmo tempo e depois agregar os resultados). O pipeline de tickets acima é um workflow, e você poderia ter desenhado o fluxograma dele antes de colocá-lo para rodar.

Um *agente* entrega o fluxo de controle ao modelo. Um ensaio de engenharia muito citado traça a linha exatamente aqui: workflows são “sistemas em que LLMs e ferramentas são orquestrados por caminhos de código predefinidos”, enquanto agentes são sistemas em que os modelos “dirigem dinamicamente os próprios processos e o uso de ferramentas”.³ O número de passos, e qual passo vem quando, não existe em lugar nenhum do seu código; o modelo escolhe à medida que avança, reagindo ao que acabou de observar. É por isso que o sistema que corrige testes parece diferente: a transcrição dele se lê como o registro da tarde de um colega, e só de longe lembra a execução de um programa.

Por baixo dos dois últimos está o mesmo átomo, que vale a pena nomear porque os padrões que você vai encontrar no capítulo 10 são montados a partir dele. O *LLM aumentado* (*augmented LLM*) é uma chamada de modelo reforçada com três coisas: *recuperação* (trazer informação relevante no momento da requisição), *ferramentas* (funções que ele pode solicitar) e *memória* (alguma forma de levar adiante o que importa). Um workflow é formado por vários desses átomos, ligados pelo seu código ao longo de um caminho fixo. Um agente é um desses átomos colocado dentro de um loop, decidindo por conta própria o que fazer a cada volta. (Um chatbot, nesse quadro, é a chamada de modelo pura, com os reforços quase todos sem uso.) O mesmo átomo, dois arranjos — guarde essa imagem e as taxonomias deixam de parecer misteriosas.

Aqui está o teste decisivo que vou usar no resto do livro. Se você consegue desenhar com segurança o diagrama do fluxo de controle antes de a requisição chegar, está diante de um workflow. Se o diagrama só pode ser desenhado

³Erik Schluntz e Barry Zhang, “Building Effective Agents”, blog de engenharia da Anthropic (2024); as duas frases são citadas do ensaio, que também dá nome ao bloco de construção *LLM aumentado*, descrito no próximo parágrafo. Um levantamento anterior — Lilian Weng, “LLM Powered Autonomous Agents”, lilianweng.github.io (2023) — descreve a mesma anatomia vista de dentro: um modelo como “cérebro”, apoiado por planejamento, memória e uso de ferramentas.

em retrospecto, depois que o modelo reagiu ao que encontrou, está diante de um agente.

É tentador tratar chatbot, workflow e agente como três caixas. A realidade é um dial (figura 1.3). Em uma ponta, o modelo gera texto e um humano decide tudo. Perto dessa mesma ponta fica o pipeline fixo (o sistema de tickets de agora há pouco), em que o código é dono de cada passo e o modelo só preenche o conteúdo. Um pouco acima, um workflow concede ao modelo uma decisão: um roteador escolhe qual ramo fixo no código trata a requisição. Mais adiante, um agente percorre livremente o loop com suas ferramentas, mas faz uma pausa para aprovação humana antes de qualquer ação com consequências — enviar o e-mail, gastar o dinheiro, apagar os dados; boa parte do que está em produção fica aqui. Na extremidade oposta, o modelo é dono do caminho de ponta a ponta, por muitos passos, sem confirmação a cada passo: capacidade máxima, *raio de impacto* (*blast radius*) máximo — o estrago que uma ação errada poderia causar antes que algo a detenha, um termo que o capítulo 17 vai tornar preciso. O que se move ao longo desse dial é uma única grandeza, quanto do fluxo de controle pertence ao modelo, e sistemas maduros escolhem uma posição deliberadamente, em vez de herdar uma de alguma demo.

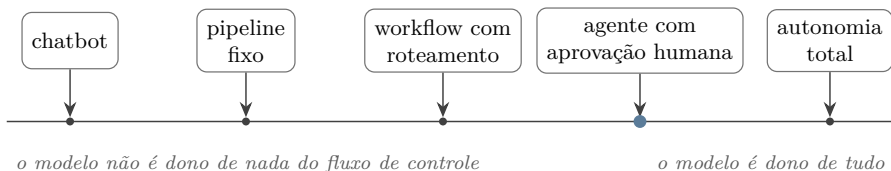


Figura 1.3: A autonomia é um dial, não um interruptor. A grandeza que varia ao longo dele é quanto do fluxo de controle pertence ao modelo. Os rótulos são paradas que vale a pena nomear; os sistemas reais costumam ficar entre elas.

Mais uma observação antes de deixarmos a taxonomia, porque ela antecipa onde este livro vai chegar. Quase nada do que vai para produção é um workflow puro ou um agente puro. O formato dominante em produção é uma casca de workflow com um ou dois passos genuinamente agênticos dentro dela: estrutura previsível onde quer que você consiga prever, autonomia nas mãos do modelo apenas nos pontos que você não consegue enumerar. Você está ajustando um dial, passo a passo, e a próxima seção trata de até onde girá-lo.

1.3 Você precisa mesmo de um agente?

Agora que você sabe distinguir um agente de um workflow, está diante da pergunta que esta seção existe para tornar respeitável: você precisa mesmo de

um? Eu a coloco tão cedo de propósito. O erro contra o qual esta área mais adverte quem está chegando é recorrer a um agente autônomo quando um único prompt, um pipeline fixo ou cinquenta linhas de código comum teriam sido mais baratos, mais rápidos e mais fáceis de depurar. Uma demo de agente é uma coisa sedutora. Você vê um modelo planejar, agir, se recuperar e terminar, e a conclusão natural é que esse poder cabe em todo lugar. Resista à conclusão por tempo suficiente para calcular o preço dela.

Comparado com uma única chamada de modelo, um agente acrescenta vários custos recorrentes, e eles vêm em pacote. Primeiro, o dinheiro: um agente faz muitas chamadas de modelo por tarefa em vez de uma, e cada chamada reenvia a conversa, que não para de crescer; assim, o custo escala duas vezes — uma no número de chamadas, outra no tamanho de cada uma. Um loop de dez passos pode custar uma ordem de grandeza a mais do que a chamada única que ele substituiu (o multiplicador é ilustrativo; o que importa é a direção). Segundo, a latência: as chamadas são sequenciais, cada uma esperando o resultado da última ferramenta, de modo que uma resposta em menos de um segundo vira muitos segundos ou minutos; se houver um humano esperando do outro lado, isso por si só pode desqualificar o projeto. Terceiro, o não determinismo: a mesma entrada pode seguir um caminho diferente e produzir uma resposta diferente a cada execução, o que quebra, sem alarde, os seus hábitos de teste, depuração e auditoria.⁴ E a superfície de falha muda de natureza, e esse é o custo que eu mais quero que você sinta: a falha limitada que a seção anterior creditou ao chatbot é exatamente aquilo de que um agente abre mão. O erro dele é uma *ação* errada, seguida de mais dez ações construídas sobre ela, com o erro se compondo ao longo do loop exatamente como prevê a aritmética da bússola.

Dois outros custos passam facilmente despercebidos quando a demo é deslumbrante. Conceder ferramentas e autonomia a um modelo amplia a sua superfície de ataque; a comunidade de segurança tem um nome para a concessão em excesso: *agência excessiva* (*excessive agency*), ou seja, mais permissões, funcionalidades ou liberdade do que a tarefa exige.⁵ O perigo também vai além do que você concede deliberadamente: um texto que o agente apenas *lê* pode direcionar o que ele faz em seguida, uma família de ataques chamada *injeção de prompt* (*prompt injection*), que o capítulo 17 examina. Por fim, o peso operacional: um chatbot precisa de um prompt, ao passo que um agente precisa, além disso, da integração das ferramentas, de gestão de contexto, recuperação de erros, condições de parada, pontos de aprovação, rastreamento (*tracing*) e avaliações. Esse maquinário ao redor é chamado de *harness*, e construí-lo é a maior parte do trabalho; um modelo brilhante em um harness

⁴Uma análise publicada sobre a confiabilidade de agentes relata taxas de sucesso em execução única em torno de 60%, que caem para cerca de 25% quando a mesma tarefa precisa dar certo em oito execuções consecutivas — números que ilustram a lacuna de consistência, não fatos permanentes. Fiddler AI, “AI Agent Failure Rate: Why 70–95% Fail in Production”, blog da fiddler.ai (2026).

⁵OWASP, “Top 10 for LLM Applications” — o item sobre Excessive Agency.

frágil falha em produção.

Portanto, o ponto de partida honesto é uma escada, que é o quarto rumo da bússola posto em prática, e a disciplina é ficar no degrau mais baixo que resolve o seu problema. O degrau de baixo é código comum: uma expressão regular, uma consulta SQL, um `if` — gratuito, instantâneo, determinístico, auditável; se a lógica pode ser totalmente especificada, não é preciso modelo nenhum. O degrau seguinte é uma única chamada de modelo bem elaborada, talvez com recuperação e alguns exemplos no prompt; uma fração surpreendente dos “precisamos de um agente” se dissolve em “precisávamos de um prompt melhor”. O degrau acima desse é um workflow: várias chamadas em caminhos que você desenhou de antemão. O degrau do topo é um agente. Cada degrau que você sobe compra adaptabilidade e paga por ela em custo, latência e imprevisibilidade. Para um problema cujos passos você consegue desenhar de antemão, um degrau mais baixo é a resposta correta em definitivo; permanecer nele é a disciplina. A figura 1.4 desenha a escada, com a adaptabilidade subindo à medida que você sobe e o custo subindo junto.

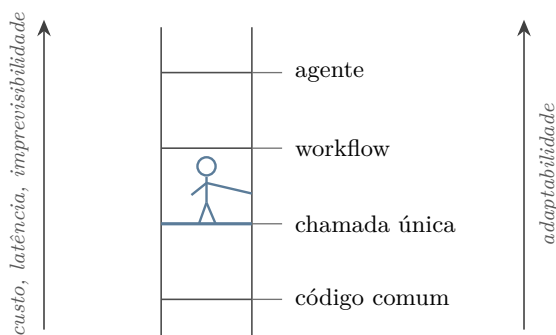


Figura 1.4: A escada de complexidade. Cada degrau que se sobe — do código comum a uma única chamada de modelo, a um workflow, a um agente — compra adaptabilidade e paga por ela em custo, latência e imprevisibilidade. A disciplina é ficar no degrau mais baixo que resolve o problema (na cor de destaque) e só subir quando um caso real provar que o degrau de baixo não alcança.

O teste prático é o da seção anterior, agora usado como apoio à decisão: você consegue desenhar o fluxograma antes de a requisição chegar? Se sim, pare em um workflow ou abaixo dele — passos conhecidos, requisitos de correção auditáveis, orçamentos apertados de latência ou de custo e tarefas simples de consultar, classificar e transformar apontam todos para baixo na escada. O que aponta para cima é o objetivo realmente aberto: os passos dependem do que o modelo descobre, e é preciso discernimento em ramificações que você não consegue enumerar. Mesmo assim, exija o primeiro rumo da bússola em forma concreta — um critério de sucesso claro, um sinal de feedback a cada

passo e um lugar sensato para um humano olhar. Um par concreto para calibrar: “encaminhe cada e-mail recebido para faturamento, suporte técnico ou vendas e redija uma primeira resposta” tem passos que dá para conhecer e alto volume, e pede um workflow; “faça este teste que está falhando passar em uma base de código que você nunca viu” tem passos impossíveis de conhecer de antemão e um verificador embutido, e é um problema genuíno de agente.

Uma armadilha merece um parágrafo só para ela, porque defende com muita eloquência o projeto errado: “pode ser que um dia precise se adaptar”. Isso é uma hipótese, e barata de testar com honestidade: construa primeiro a versão mais simples, meça-a com casos reais e só suba a escada quando puder apontar uma classe de entradas em que o sistema simples comprovadamente falha e confirmar que um agente de fato se sai melhor. A armadilha inversa é o *agent-washing*: como a palavra vende, pipelines de roteiro fixo são rebatizados de agentes. A pergunta que desmonta as duas é o teste decisivo que você já tem, feito ao sistema que está na sua frente, seja qual for o rótulo: é o modelo que decide o próximo passo, ou é o código? O capítulo 14 retoma essa decisão com a bagagem de todos os capítulos intermediários; o capítulo 16 mostra como fazer a medição.

Um agente, então, é um custo que você paga por uma adaptabilidade que consegue nomear. Se não consegue nomear a adaptabilidade, guarde o seu dinheiro. E se consegue — se os passos realmente não podem ser desenhados de antemão e existe um sinal de verificação — então um agente é a ferramenta certa, e os próximos capítulos tratam de construí-lo bem. Antes, porém, você precisa entender o motor que vamos colocar em loop: o capítulo 2 é sobre o que um modelo de linguagem realmente é.

Capítulo 2

O motor: como funcionam os modelos de linguagem

Todo sistema deste livro, do chatbot mais simples a uma frota de agentes que cooperam entre si, é construído em torno do mesmo componente: um grande modelo de linguagem. Dá para usar um deles de forma produtiva sem saber nada do que há lá dentro, do mesmo jeito que dá para dirigir um carro sem nunca abrir o capô. Construir um agente é outro tipo de relação com a máquina. Você vai colocar esse componente em um loop, entregar ferramentas a ele e deixá-lo tomar decisões enquanto você está em outro lugar; quando ele se comportar de um jeito estranho, “é mágica” não vai ajudar a localizar o bug. Este capítulo monta o modelo mental mínimo de que quem constrói agentes precisa, com a intuição em primeiro lugar e nenhuma matemática além da aritmética. Começamos pelo que o modelo é, no fundo (um preditor do próximo token), e passamos depois à sua memória de trabalho (os tokens e a janela de contexto), ao modo como ele escreve (a amostragem, e por que duas execuções diferem), ao modo como você o direciona (a escrita de prompts), à ponte entre o texto dele e a ação real (a saída estruturada e a chamada de função) e, por fim, aos modos como ele falha com regularidade. Os capítulos seguintes se apoiam em cada uma dessas ideias; nenhuma delas exige que você treine coisa alguma.

2.1 LLMs como preditores do próximo token

Complete esta frase: “A capital da França é —”. A palavra estava na sua cabeça antes de você decidir procurá-la. Você não consultou um atlas mental, não pesou cidades candidatas nem raciocinou sobre geografia; uma vida inteira de leitura simplesmente fez *Paris* aparecer. Guarde esse reflexo, porque ele é a intuição mais precisa que existe para o que um grande modelo de

linguagem faz. O modelo completa texto do jeito que você acabou de fazer; a diferença é que ele não tem nenhum outro modo de funcionar.

Mecanicamente, um LLM é uma função: ele recebe um trecho de texto e produz, para cada *token* possível, a probabilidade de que ele seja o próximo. Um token é um pedaço de texto tirado de um vocabulário fixo — muitas vezes uma palavra, às vezes um pedaço de uma palavra ou um sinal de pontuação; em média, pense em cerca de três quartos de uma palavra em inglês. Para escrever qualquer coisa, o sistema ao redor escolhe um token dessa lista ordenada, acrescenta-o ao texto e roda o modelo de novo sobre o texto agora mais longo, um token de cada vez, repetidamente, até que uma condição de parada dispare. Esse esquema se chama *geração autorregressiva* (cada novo token é condicionado a tudo o que veio antes dele, prompt e saída igualmente), e uma propriedade dele vai nos acompanhar pelo livro inteiro: o modelo nunca faz um rascunho da resposta com antecedência. Todo o seu método é um palpite local, repetido; ele não guarda nenhum plano nem roteiro de reserva (figura 2.1). A próxima seção examina a fundo os tokens e a memória de trabalho do modelo; a seguinte mostra como um token é escolhido e por que a escolha varia.

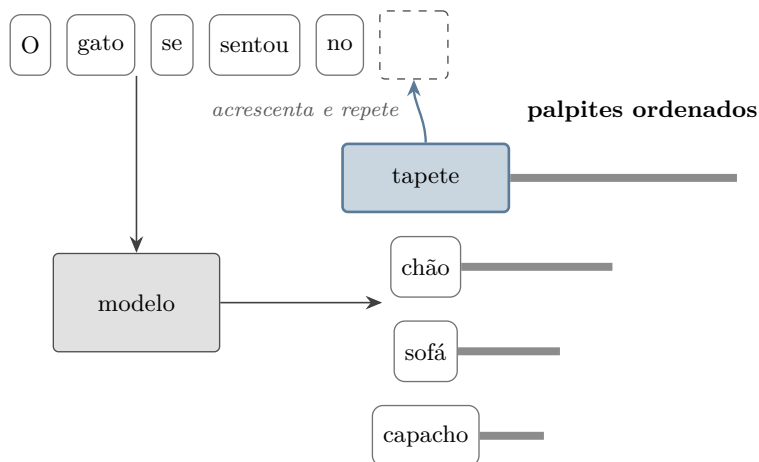


Figura 2.1: A geração autorregressiva, todo o método do motor. O texto até aqui entra no modelo, que devolve uma lista ordenada de palpites para o próximo token; um deles é escolhido e acrescentado ao texto, e o modelo roda de novo sobre a entrada, agora mais longa. O token escolhido e o seu caminho de volta estão na cor de destaque — não há nenhum plano guardado de reserva, só este palpite local, repetido.

Onde mora essa capacidade de adivinhar? Nos *parâmetros* do modelo, também chamados de *pesos* — bilhões de números, fixados durante o treinamento, que determinam como o texto de entrada flui pela rede neural para

produzir aquelas probabilidades. A imagem que acho mais útil é esta: os pesos são uma destilação comprimida, e com perda, dos padrões de tudo o que o modelo leu durante o treinamento — gramática, fatos, expressões idiomáticas, estilo de código, a forma dos argumentos. “Com perda” é a expressão que importa. O modelo absorveu a forma do que leu e não guardou nenhum arquivo do texto em si, e é por isso que pedir a ele que cite uma fonte palavra por palavra, ou que lembre um número exato, é uma aposta; voltamos às consequências no fim deste capítulo.

Duas atividades precisam ficar bem separadas, porque confundi-las produz alguns dos mal-entendidos mais comuns sobre esses sistemas. O *treinamento* é o processo único e caríssimo de encontrar bons valores para os pesos: o modelo vê texto, precisa prever o próximo token e recebe um pequeno ajuste toda vez que erra, ao longo de mais texto do que uma pessoa conseguiria ler em mil vidas. Ele é feito por quem fabrica o modelo, em hardware especializado, antes de você sequer ver a coisa. A *inferência* é todo uso posterior: os pesos estão congelados, e o modelo simplesmente passa a sua entrada por eles. Tudo o que você faz com um modelo — cada conversa, cada passo de agente — é inferência. Daí decorrem duas consequências, e você vai projetar em torno das duas o tempo todo. Primeiro, o modelo não aprende com as suas conversas; nada do que você digita muda um único peso, e o que parece memória dentro de uma sessão é só a transcrição, cada vez maior, sendo relida a cada chamada. Segundo, o conhecimento embutido no modelo para em uma *data de corte do treinamento* (o ponto a partir do qual ele não viu mais dados). Os acontecimentos posteriores ao corte simplesmente não existem para ele, e o modelo não vai anunciar a lacuna de forma confiável; ele responde a partir de um conhecimento desatualizado com exatamente a mesma convicção com que fala do que é recente. Tudo o que depende do momento precisa chegar no prompt, ou ser buscado por uma ferramenta.

O próprio treinamento acontece em etapas, e as etapas explicam um enigma em que você tropeçaria de outra forma. A primeira etapa, e a mais cara, é o *pré-treinamento*: predição pura do próximo token sobre um corpus vasto e variado — páginas da web, livros, código. O que sai dela é um *modelo base*: fluente, com conhecimento amplo e estranhamente inútil como assistente. O capítulo 1 mencionou, de passagem, que um modelo assim pode responder a uma pergunta com mais perguntas, e prometeu que este capítulo mostraria por quê. Você já tem todas as peças da explicação; monte-as antes de continuar a leitura. Um modelo base só continua texto, então, quando você lhe entrega “Quais são três maneiras de deixar esta consulta mais rápida?”, ele procura uma continuação plausível — e, no texto do mundo real que ele estudou, uma pergunta muitas vezes vem seguida de outras parecidas. Uma lista de novas perguntas é uma resposta perfeitamente provável. O modelo base está fazendo o seu trabalho com perfeição; o trabalho é continuar.

O *pós-treinamento* (*post-training*) é a etapa seguinte, que transforma esse continuador bruto no assistente que você de fato conhece, e é a ele que se

referia o “os modelos aprenderam a receber instruções” do capítulo 1. Dois movimentos importam para o nosso propósito. O *ajuste por instruções* (*instruction tuning*) treina o modelo com pares selecionados de instrução e boa resposta, e lhe ensina que uma pergunta merece uma resposta, e não mais perguntas. O *ajuste por preferências* (*preference tuning*) — cuja forma mais conhecida é o *RLHF*, aprendizado por reforço com feedback humano — põe pessoas (ou modelos treinados para imitar o julgamento delas) para ordenar respostas concorrentes e ajusta os pesos na direção do tipo de resposta que as pessoas preferem: mais útil, mais honesta, mais educada. É do pós-treinamento que vem a persona de assistente que você conhece, e ele é uma demão de tinta relativamente barata sobre o enorme núcleo pré-treinado.

Tenha a mesma clareza sobre o que o pós-treinamento não dá. Ele não acrescenta conhecimento novo e não elimina a tendência a inventar. Uma formulação memorável, de uma aula muito assistida sobre o assunto: o pós-treinamento não impede o modelo de sonhar; ele dá aos sonhos o formato de sonhos de um assistente prestativo.¹ Debaixo das boas maneiras, o motor é o mesmo: um preditor do próximo token, que otimiza a continuação plausível. Plausível e verdadeiro estão correlacionados. A correlação existe porque o texto de treinamento foi escrito, em sua maior parte, por pessoas descrevendo um mundo real — mas os dois se separam exatamente onde falta ao modelo o fato, e nesse momento o maquinário produz a sequência de aparência mais plausível em vez de parar. Fluente primeiro, correto depois.

Para quem constrói agentes, o corolário que mais quero que você guarde é que o conhecimento do modelo mora em dois lugares, de confiabilidade desigual. O conhecimento *paramétrico* fica nos pesos: congelado na data de corte, aproximado, impossível de citar. O conhecimento *de contexto* é qualquer texto que esteja diante do modelo neste momento: atual, inspecionável, verificável. O modelo é bem mais confiável quando os fatos de que precisa estão no seu contexto do que quando precisa pescá-los nos pesos, e essa assimetria é a semente de metade da engenharia deste livro. A recuperação, as ferramentas e toda a disciplina da engenharia de contexto, na Parte III, são, no fundo, maneiras de tirar fatos que carregam peso do primeiro lugar e levá-los ao segundo.

Este é, portanto, o motor: excelente em trabalho com formato de linguagem — redigir, reescrever, resumir, traduzir, explicar, transformar código — e pouco confiável como banco de dados ou calculadora, com um dial de convicção travado para sempre no máximo. Um agente vai pedir a esse motor que tome decisões, muitas em sequência, sobre um mundo que ele só consegue ver pelo buraco da fechadura do seu contexto. O buraco da fechadura é o nosso próximo destino.

¹A imagem do “sonhar” é de Andrej Karpathy, das suas palestras e comentários sobre grandes modelos de linguagem, que circularam amplamente (p. ex. “Intro to Large Language Models”); aparece aqui como paráfrase, não como citação literal.

2.2 Os tokens e a janela de contexto

Tudo o que o modelo sabe em um dado momento chega por esse buraco da fechadura, então é melhor medi-lo. O capítulo 1 precisou de pouco mais do que uma definição da *janela de contexto*: o trecho limitado de texto que um modelo consegue considerar em uma única chamada. Esta seção completa o quadro, e o ponto de partida é a unidade em que a janela é medida, porque quase todo limite que você vai encontrar ao construir — o teto do que o modelo consegue ver, a conta no fim do mês, a pausa antes de uma resposta começar — é expresso em tokens. A seção anterior ofereceu uma versão de guardanapo: um token é um pedaço de texto, em média cerca de três quartos de uma palavra em inglês.² A versão de guardanapo esconde um problema de projeto real, e a solução desse problema explica vários comportamentos que, sem ela, pareceriam pura burrice.

O problema de projeto é este. Um modelo precisa de um vocabulário fixo: um cardápio finito de pedaços com os quais toda entrada e toda saída possível serão montadas. Suponha que o cardápio fosse de palavras inteiras. O vocabulário incha — as formas do inglês *love*, *loves*, *loved*, *lovingly* precisam, cada uma, de uma entrada própria — e o primeiro erro de digitação, nome inventado ou palavra criada depois do treinamento não tem entrada nenhuma. Suponha, em vez disso, que o cardápio fosse de caracteres isolados. Agora nada fica sem representação e o vocabulário é minúsculo, mas todo texto vira uma sequência enorme de unidades quase sem significado. Os modelos em produção se acomodam no meio-termo entre os dois, chamado de *tokenização em subpalavras*: um vocabulário fixo de pedaços, muito maior do que um alfabeto e muito menor do que um dicionário com todas as formas de todas as palavras, em que as palavras comuns viajam como um único token e as mais raras são montadas a partir de pedaços, de modo que *annoyingly* pode virar *annoying* + *ly*, e *unhappiness* pode virar *un* + *happi* + *ness*.³ As sequências frequentes são representadas de forma barata; qualquer sequência pode ser representada, recorrendo a pedaços menores.

A máquina que faz os cortes, o *tokenizador*, merece um parágrafo curto, porque é muito mais simples do que as pessoas imaginam. É uma tabela de consulta fixa e determinística, construída uma única vez junto com o modelo e congelada pela vida inteira dele. Na entrada, ela converte o seu texto em uma lista de IDs inteiros; na saída, converte de volta em texto os IDs que o modelo escolheu. Repare no que isso implica: a rede nunca vê os seus caracteres. Do

²A regra dos três quartos de palavra (o que equivale a cerca de quatro caracteres por token em inglês) vem da nota da central de ajuda da OpenAI “What are tokens and how to count them?”; é uma média aproximada para prosa em inglês, não uma constante.

³A documentação de tokenizadores da Hugging Face (“Summary of the tokenizers”) descreve os algoritmos de subpalavras padrão — BPE, WordPiece, Unigram — e os ilustra com divisões como essas. Para um mergulho prático, a aula de Andrej Karpathy “Let’s Build the GPT Tokenizer” constrói um tokenizador do zero e percorre os comportamentos estranhos que ele causa.

lugar onde o modelo está, o mundo é uma sequência de pedaços numerados, e o que quer que aconteça *dentro* de um pedaço foi apagado antes de o modelo entrar em cena.

Esse apagamento pode ser testado, e recomendo que você mesmo faça o teste. Pergunte a um modelo quantas vezes a letra **r** aparece em alguma palavra comum, ou peça a ele que solete uma palavra de trás para a frente. Os modelos tropeçaram exatamente nesse tipo de pergunta durante anos, às vezes de forma cômica, na mesma conversa em que resolviam um problema genuinamente difícil. O padrão deixa de parecer absurdo no momento em que você tem o tokenizador em mente: a palavra chegou como um ou dois pedaços opacos, e as letras dentro de um pedaço são exatamente a informação que a tabela de consulta jogou fora. Para contá-las, o modelo precisa reconstruir a grafia a partir do que quer que tenha absorvido sobre aquela palavra durante o treinamento — uma tarefa de memória, e pouco confiável. (Quando você ler isto, o modelo que testar pode muito bem acertar; lacunas assim são remendadas pelo treinamento, ou o modelo recorre discretamente a uma ferramenta. O mecanismo que a falha expõe continua lá embaixo.) Os dígitos contam a mesma história: em pelo menos um tokenizador muito usado, 520 é um único token, enquanto 521 se divide em $5 + 21$,⁴ de modo que o modelo faz contas sobre pedaços que ignoram o sistema de numeração — um dos motivos pelos quais a aritmética de cabeça dele é mais frágil do que a sua fluência sugere. A lição vale de forma geral: o mundo do modelo tem uma granulação, e o grão é o token.

Três consequências da tokenização importam mais para quem constrói do que o funcionamento interno dela. Primeiro, a proporção de tokens por palavra varia com o conteúdo: código, números e dados estruturados se tokenizam com menos eficiência do que a prosa, então uma página de JSON custa mais tokens do que uma página de ensaio. Segundo, a maioria dos tokenizadores foi treinada predominantemente em inglês, então o mesmo significado pode custar várias vezes mais tokens em outra língua — uma análise constatou que o birmanês chega a cerca de oito vezes a contagem de tokens do inglês equivalente, um número que cito como ilustração da disparidade, não como constante⁵ —, o que tem consequências diretas de custo e de equidade se os seus usuários escrevem em muitas línguas. Terceiro, distinções em que você nunca pensaria são reais para o tokenizador: a mesma palavra com e sem um

⁴Quais números se dividem é um fato arbitrário do treinamento de cada tokenizador. No tokenizador byte-pair da geração GPT-2/GPT-3, por exemplo, 520 é um token, enquanto 521 vira $5 + 21$; tokenizadores posteriores tendem a manter inteiros os números curtos, mas ainda dividem os mais longos de forma irregular. O que importa é que alguns números se dividem, e o modelo não consegue ver dentro de um pedaço.

⁵Aleksandar Petrov et al., “Language Model Tokenizers Introduce Unfairness Between Languages”, NeurIPS 2023 (arXiv:2305.15425), que mede disparidades de até cerca de quinze vezes em uma série de tokenizadores; o número de cerca de $8\times$ para o birmanês é uma ilustração conservadora de quão grande a diferença pode ficar para línguas pouco representadas no treinamento dos tokenizadores.

espaço à frente, ou com e sem maiúscula, pode ser um token completamente diferente. O hábito prático que decorre das três: quando uma contagem importa, meça o seu texto real com o tokenizador ou o endpoint de contagem do próprio fornecedor. Cada família de modelos corta o texto de um jeito, então uma contagem emprestada de uma é só uma estimativa para outra. (Reparo que quase todo número desta seção chega embrulhado em um “cerca de”. Com tokenizadores, “cerca de” é o tom honesto.)

Agora, a janela em si. A imagem que vou usar para ela, aqui e no resto do livro, é uma mesa de trabalho: a superfície de trabalho do lado do modelo no buraco da fechadura, sobre a qual tudo o que o modelo consulta durante uma única chamada precisa caber ao mesmo tempo. Espalhados por essa mesa, em uma chamada típica de agente, estão as suas instruções permanentes, a conversa inteira até ali, as definições de todas as ferramentas que o agente pode usar, o texto completo de cada resultado de ferramenta que já voltou, os documentos buscados para a tarefa — e a resposta que o modelo está escrevendo, que vai ocupando a mesma mesa à medida que cresce. Um único orçamento de tokens cobre tudo isso. Não há gaveta, nem prateleira, nem uma segunda mesa. Se um fato está na mesa durante esta chamada, o modelo pode usá-lo; se está em qualquer outro lugar do universo, então, enquanto durar esta chamada, ele não existe.

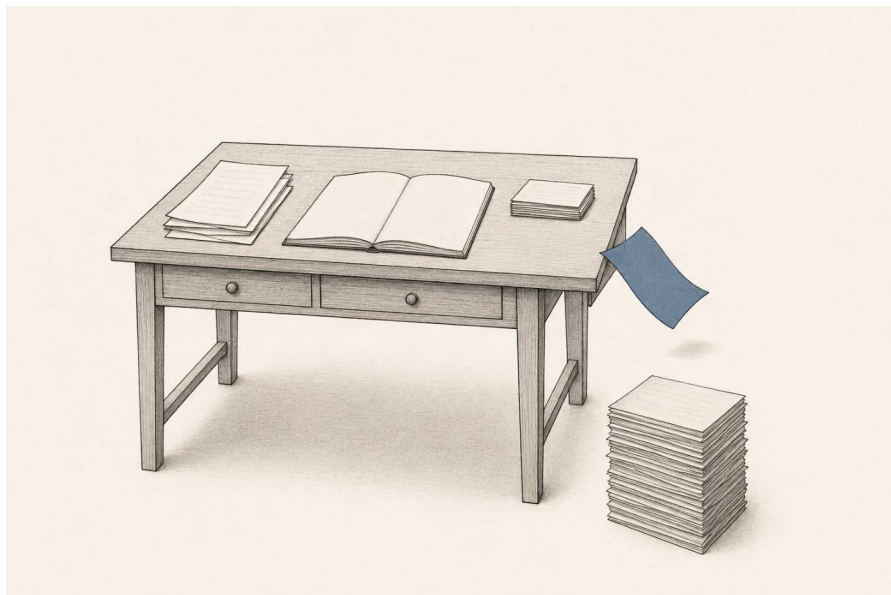


Figura 2.2: A janela de contexto como uma única mesa finita: tudo o que o modelo pode usar neste momento precisa caber fisicamente na sua superfície.

A mesa também fica limpa entre uma chamada e outra, e aqui a metá-

fora precisa da sua fronteira honesta: a sua memória de trabalho atravessa a tarde, enquanto a do modelo não sobrevive até a próxima requisição. Uma API de LLM é *sem estado* (cada chamada começa do nada), e isso dá forma operacional ao que a seção anterior disse sobre memória: para continuar uma conversa, o software ao redor põe a transcrição inteira de volta na mesa a cada turno, como você talvez releia uma conversa inteira de e-mails antes de responder à mensagem mais recente. O hábito é inofensivo com três mensagens e ruinoso com trezentas. Um esboço, com números inventados mas realistas: instruções permanentes, 500 tokens; conversa até aqui, 8.000; dois documentos buscados, 40.000; o último resultado de ferramenta, 3.000. O modelo lê mais de cinquenta mil tokens antes de escrever uma palavra, você paga por cada um deles, e no próximo turno a pilha é ainda maior, porque este turno passou a fazer parte do histórico. (Os fornecedores suavizam o custo de reler uma pilha que não mudou com cache, uma das principais alavancas do capítulo 19.)

O que acontece quando a pilha já não cabe? Nada de gracioso. Dependendo da API e do software no meio do caminho, a requisição pode ser rejeitada de cara com um erro; ou a entrada pode ser truncada — cortada até caber, em geral descartando os turnos mais antigos, muitas vezes sem avisar ninguém —; ou um framework no meio do caminho pode resumir o histórico antes de enviá-lo. O erro explícito é o despecho gentil. O truncamento silencioso é o traçoeiro: a chamada dá certo, a resposta volta fluente, e o modelo estava raciocinando sobre uma transcrição amputada da qual as suas instruções originais podem ter caído sem barulho. Visto de fora, parece um modelo que ficou mais burro da noite para o dia, e consome horas de depuração na mesma medida. Quando uma sessão longa começar a “esquecer”, verifique se houve estouro antes de pôr a culpa no modelo. (Um parente próximo dessa surpresa: entrada e saída dividem a janela, mas muitos modelos limitam separadamente uma única resposta a um número bem menor. Uma resposta que para no meio da frase em geral bateu no teto de saída.)

Mais uma propriedade da mesa, que o capítulo 1 já deixou escapar: mesmo o que cabe é usado de forma desigual. Em entradas longas, os modelos recuperam o material perto do começo e perto do fim bem melhor do que o material enterrado no meio — a bússola chamou um fato assim de meio esquecido, e a literatura de pesquisa chama o efeito, com propriedade, de “perdido no meio” (*lost in the middle*).⁶ Uma mesa maior, portanto, resolve menos do que parece: as bordas continuam nítidas enquanto o meio cresce. Aqui está a base mecânica do segundo rumo do livro, a escassez de contexto, e o ofício que ela impõe — decidir o que merece um lugar na mesa — ocupa a Parte III, a começar pelo capítulo 7.

⁶Nelson F. Liu et al., “Lost in the Middle: How Language Models Use Long Contexts”, *Transactions of the Association for Computational Linguistics* (2024). A acurácia em tarefas de contexto longo foi maior quando o trecho relevante estava no início ou no fim da entrada e caiu quando ele estava no meio.

Mesmo quando tudo cabe e fica onde a recuperação é boa, a pilha nunca é de graça. Os tokens são a unidade da sua conta nos dois sentidos, entrada e saída. São também a unidade da sua espera: antes de emitir o primeiro token da resposta, o modelo precisa ler a pilha inteira — uma fase chamada *prefill* —, e, na arquitetura padrão, o custo dessa leitura cresce depressa com o comprimento, aproximadamente na ordem do seu quadrado. Um contexto inchado faz o agente demorar mais para começar a responder e sair mais caro em cada uma das chamadas, quer o material extra ajude, quer não. Por isso, a janela é um teto, não uma meta; a pergunta útil é qual é a menor mesa que de fato comporta a sua tarefa.

Para um chatbot, esses são fatos de fundo. Para um agente, são a previsão do tempo de todo dia: o loop acrescenta coisas à mesa a cada passo — raciocínio, chamada de ferramenta, resultado de ferramenta, de novo e de novo —, e ninguém faz a curadoria da pilha se você não construir a curadoria. Vinte passos depois, a transcrição está carregada de listagens de diretório desatualizadas e de páginas despejadas que só em parte vêm ao caso, a instrução que realmente importa está no meio da pilha, onde a recuperação cai, e cada novo passo lê, e cobra, todo o acúmulo. A mesa agora está medida. O que resta da mecânica é a escrita em si: como o modelo escolhe um token entre os seus palpites ordenados, e por que duas execuções idênticas do seu agente não vão se comportar de forma idêntica. É o que vem a seguir.

2.3 Como o texto é gerado

Entregue ao modelo o fragmento “O gato se sentou no” e, como descreveu a abertura deste capítulo, volta uma lista ordenada: uma probabilidade para cada token do vocabulário. Digamos que o topo dessa lista traga **tapete** com 0,42, **chão** com 0,18, **sofá** com 0,13 e **capacho** com 0,09, com as últimas migalhas de probabilidade espalhadas por milhares de azarões. (Os números são inventados; a forma é típica.) Agora, a pergunta em torno da qual esta seção gira, e gostaria que você se comprometesse com uma resposta antes de continuar a leitura: que palavra o modelo escreve? A resposta que parece óbvia é **tapete** — fique com o melhor palpite; por que uma máquina faria outra coisa? No entanto, para a maioria dos sistemas que você vai chamar na vida, essa resposta está errada. Por padrão, o maquinário joga dados.

O passo em questão — transformar uma lista ordenada em um único token escolhido — se chama *decodificação*, e a estratégia óbvia tem nome: *decodificação gulosa*, que sempre fica com o token mais provável. A decodificação gulosa é determinística, barata e soa como a escolha responsável. Ela também produz texto com um mal bem documentado: uma prosa sem relevo, estranhamente repetitiva, que pode travar em loops, a mesma frase reaparecendo sem parar.⁷ O motivo é contraintuitivo o bastante para valer uma explicação.

⁷Ari Holtzman et al., “The Curious Case of Neural Text Degeneration”, ICLR 2020.

A escrita humana vive correndo pequenos riscos — um adjetivo inesperado, uma expressão nova —, enquanto um decodificador que nunca arrisca escorrega para o canto mais inosso da língua e fica rodando ali. A palavra mais segura no nível local, escolhida todas as vezes, forma uma passagem medíocre no nível global.

A alternativa, e o padrão em quase todo lugar, é a *amostragem*: sortear o próximo token, com pesos dados pelas probabilidades do modelo. **tapete** ganha o sorteio 42 vezes em cada cem; **capacho** tem a sua vez umas nove vezes em cada cem. A imagem a guardar é a de uma *loteria ponderada*, realizada uma vez por token: o modelo imprime as chances, e o decodificador tira um bilhete. Toda resposta fluente e de som natural que você já admirou foi montada assim, um sorteio de cada vez. E uma propriedade dos seus futuros agentes está sendo fixada aqui, antes de existir qualquer agente: a variabilidade faz parte do projeto. Duas requisições idênticas podem tirar bilhetes diferentes, e, no momento em que um sorteio difere, os dois textos se separam para sempre, porque cada token seguinte é condicionado aos anteriores.

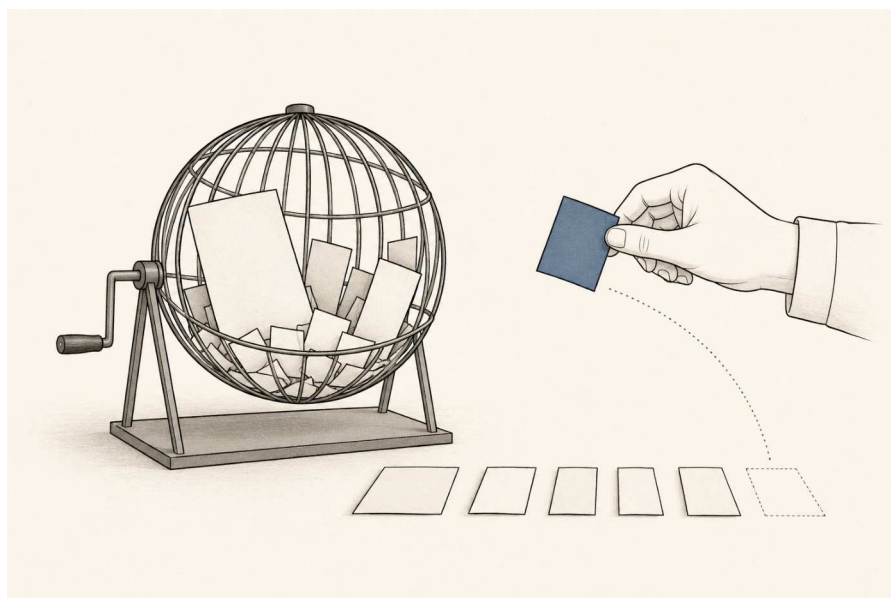


Figura 2.3: A amostragem como uma loteria ponderada, realizada uma vez por token. O modelo imprime as chances e o decodificador tira um bilhete; o favorito em geral ganha, mas nem sempre, e cada sorteio ocupa o seu lugar na frase escrita até ali.

Entre o guloso e o desvairado há botões de ajuste, e todos fazem uma única

O artigo documentou como a decodificação do tipo maximização degenera em repetição e introduziu a amostragem por núcleo (top- p) como remédio.

coisa: remodelar ou aparar a lista ordenada antes do sorteio. Todos atuam depois da predição: a lista ordenada já está fixada, e os botões governam só a maneira de escolher dentro dela. A *temperatura* é o dial de nitidez. Girada para baixo, ela aumenta a distância entre os favoritos e os azarões até que, em zero, o primeiro token ganha todos os sorteios e você está de volta à decodificação gulosa; girada para cima, ela nivela o campo, dá chances reais aos azarões e acaba dissolvendo o texto em ruído. Repare no trade-off que o dial faz: temperatura baixa rende consistência, temperatura alta rende variedade, e a correteude não está em nenhuma das pontas — uma resposta errada pode ser sorteada com toda a convicção em qualquer ajuste.

Os outros dois botões comuns aparam o campo em vez de remodelá-lo. O *top-k* mantém só os k tokens de maior probabilidade e descarta o resto — simples, mas k é um tamanho fixo de grupo, e o número de continuções genuinamente plausíveis varia enormemente de um momento para outro. O *top-p* (também chamado de *amostragem por núcleo*, em inglês *nucleus sampling*) corrige isso mantendo o menor conjunto de tokens do topo cujas probabilidades somam pelo menos p : depois de “A capital da França é”, o grupo pode ter um único token; depois de “Minha comida favorita é”, pode ter dezenas. O grupo se dimensiona pela própria certeza do modelo, e é por isso que o *top-p* é um padrão comum. A orientação prática é curta. Escolha o ajuste pela tarefa — temperatura baixa para tudo o que um programa vai analisar ou um teste vai verificar, mais alta para brainstorming e rascunhos — e mexa em um botão de cada vez, já que vários deles espremem a mesma distribuição.

A geração também precisa terminar, e termina de uma de três maneiras. O modelo pode emitir um *token de fim de sequência* especial — o treinamento o ensinou a dizer “terminei” na forma de vocabulário —, que é o caminho feliz. Um teto rígido de comprimento máximo pode cortar a saída no meio de um pensamento; a seção anterior encontrou esse caso como a resposta que para no meio da frase. Ou uma *sequência de parada* fornecida por quem chama (uma string que você designa de antemão, muitas vezes um delimitador) encerra a geração no momento em que é produzida. Um hábito para criar desde já: as APIs em geral informam qual dessas coisas encerrou a resposta. Leia esse campo. Uma resposta truncada pelo teto de comprimento parece exatamente uma falha do modelo, e na verdade é um orçamento que você definiu.

Agora, a pergunta que a seção anterior deixou no ar: por que duas execuções idênticas do seu agente vão diferir. A primeira fonte de variação está à vista de todos. A amostragem é uma loteria, então qualquer temperatura acima de zero significa que o mesmo prompt pode tirar bilhetes diferentes, se bifurcar e nunca mais se reencontrar. Essa fonte é intencional e está inteiramente sob o seu controle: ponha a temperatura em zero, e o sorteio se torna determinístico. Em teoria.

Aqui vai um exercício de previsão para a intuição que você tem agora. Suponha que um mesmo prompt seja enviado mil vezes a um modelo hospedado, com temperatura zero (decodificação gulosa, sem loteria). Quantas respostas

distintas deveriam voltar? A aritmética na sua cabeça diz uma. Quando uma equipe de pesquisa fez de fato esse experimento, obteve oitenta. As mil respostas foram idênticas nos primeiros 102 tokens; no token 103 — a frase tinha chegado à cidade natal de um personagem biográfico —, 992 delas a formularam de um jeito e 8 de outro, e a partir dessa bifurcação as transcrições se espalharam.⁸ A explicação popular culpa os erros de arredondamento do hardware paralelo, e está incompleta. A explicação mais completa envolve uma propriedade chamada *invariância ao lote* (*batch invariance*), ou melhor, a ausência dela. Um endpoint hospedado atende muitos clientes ao mesmo tempo e agrupa as requisições deles em lotes, por eficiência; o tamanho do lote sobe e desce com o tráfego; e as rotinas numéricas de baixo nível dentro do modelo podem devolver resultados minimamente diferentes com tamanhos de lote diferentes. Minimamente basta. Quando dois tokens estão quase empatados no topo da lista ordenada, uma oscilação na, digamos, sexta casa decimal decide o vencedor, a decodificação gulosa o escolhe fielmente, e o texto se bifurca. Do seu lado da API, a entrada decisiva é quantos desconhecidos estavam consultando os mesmos servidores no mesmo instante — uma entrada que você não consegue ver, controlar nem reproduzir. Existem versões invariantes ao lote dessas rotinas, e elas de fato resolvem o problema, ao custo de velocidade de atendimento; no momento em que escrevo, são a exceção. Trate então a temperatura zero como uma preferência forte, respeitada quase sempre — e lembre que “quase sempre”, sorteado a cada token de cada passo, deixa espaço para surpresas.

O que quem constrói deve fazer com tudo isso? Três hábitos. Primeiro, abra mão da reprodutibilidade exata da string como fundamento: escreva testes que verifiquem propriedades da saída — ela é analisável, cita o cliente certo, passa no verificador —, porque um teste que exige igualdade byte a byte vai ficar instável de maneiras que desperdiçam semanas suas. Segundo, espere que a falha que você viu uma vez resista a ser reproduzida quando você quiser; isso é um fato da infraestrutura, e o capítulo 15 constrói as ferramentas de gravação que tornam essas falhas depuráveis mesmo assim. Terceiro, sinta a consequência com formato de agente: um agente roda essa loteria a cada token de cada passo, e a saída de cada passo vira a entrada do seguinte, então a variação entre execuções se compõe ao longo do loop — a mesma tarefa, caminhos diferentes, ambos plausíveis. A resposta a isso que melhor envelheceu na área é tornar o resultado verificável, em vez de tornar a saída idêntica: o rumo da verificabilidade do capítulo 1, chegando agora com a sua justificativa mecânica.

A máquina escreve, portanto, por loteria, e a loteria explica a variedade.

⁸Horace He et al., “Defeating Nondeterminism in LLM Inference”, Thinking Machines Lab (2025). O experimento: 1.000 respostas com temperatura zero ao prompt “Tell me about Richard Feynman” (fale sobre Richard Feynman) produziram 80 saídas distintas, idênticas até o token 102 e divergindo pela primeira vez na expressão do local de nascimento (“Queens, New York” versus “New York City”). Os números ilustram o fenômeno, não são uma constante da natureza.

A próxima questão é o direcionamento. Se os pesos estão congelados e os dados são indiferentes, a única alavanca que sobra é o texto que você põe na mesa — o que dizer, em que ordem e o que mostrar por meio de exemplos. Essa alavanca é a próxima seção.

2.4 Fundamentos da escrita de prompts

Conte as alavancas que você realmente tem sobre essa máquina. Os pesos estão congelados; nada do que você faz em tempo de inferência vai mudá-los. Os botões de amostragem da seção anterior ajustam a maneira como um token é sorteado da lista ordenada, nunca o que o modelo previu de saída. Todo o resto se reduz a um único instrumento: o que você escolhe escrever. Esse texto é o *prompt*, e o ofício de escrevê-lo bem — a *engenharia de prompt*, um nome pomposo para uma prática que é em parte escrita e em parte experimento — é a competência básica de direcionamento sobre a qual tudo o que é agêntico se apoia. Os seus fundamentos cabem nesta seção, e vale a pena relê-los mesmo que você escreva prompts há anos: um agente monta um prompt novo a cada passo do seu loop, então uma fraqueza aqui é uma fraqueza repetida em tudo o que você construir depois.

Comece pela anatomia. Os modelos de chat modernos recebem uma lista de *mensagens*, cada uma marcada com um *papel*, e três papéis são quase universais (os nomes exatos e os formatos de transmissão variam conforme o fornecedor; trate qualquer sintaxe específica como detalhe do fornecedor). A mensagem *system* (de sistema) leva as instruções permanentes: quem o modelo deve ser, o que pode e o que não pode fazer, e o formato que as respostas precisam ter. Ela é escrita uma vez e persiste por todos os turnos (o software ao redor a reenvia a cada chamada — o modelo em si, lembre-se, não retém nada). A mensagem *user* (do usuário) leva o pedido do momento — esta pergunta, este ticket, este arquivo. E as mensagens *assistant* (do assistente) são as respostas do próprio modelo, que se acumulam como o histórico da conversa; um detalhe útil é que você também pode escrever mensagens de assistente, pondo palavras na boca do modelo como forma de mostrar, em vez de descrever, como é uma boa resposta. A minha regra de triagem: se uma instrução precisaria ser repetida em todo turno, ela pertence à mensagem de sistema; se é específica desta requisição, pertence à mensagem do usuário. Misturar as duas coisas cobra o seu preço depois — uma regra repetida em cada mensagem do usuário só sobrevive enquanto todos os que chamam o modelo se lembrarem de repeti-la. A figura 2.4 mostra os três papéis e a ausência de estado que reenvia a pilha inteira a cada chamada.

A maior alavanca de todas é a clareza, e depois de duas seções de mecânica você consegue ver por que ela funciona. O modelo continua texto; uma instrução vaga deixa a lista ordenada larga, e a loteria passeia à vontade por todo o espaço que você deixa aberto. Uma instrução precisa estreita o campo

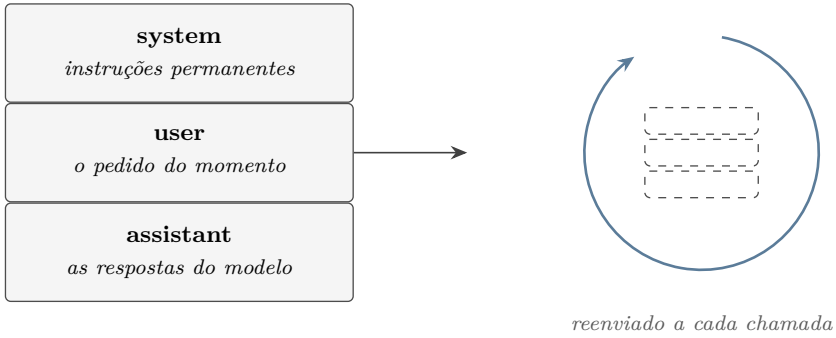


Figura 2.4: A anatomia de um prompt de chat. As instruções permanentes vão na mensagem *system*, o pedido do momento na mensagem *user*, e as respostas do próprio modelo se acumulam como mensagens *assistant*. Como o modelo não retém nada entre uma chamada e outra, o software ao redor reenvia a pilha inteira a cada turno (o loop na cor de destaque) — o que parece memória é só a transcrição sendo relida.

antes de os dados rola-rem. Na prática, isso significa preferir restrições verificáveis a adjetivos: “de três a cinco frases” é melhor do que “seja breve”, e “no máximo cinco tópicos, cada um com menos de doze palavras” é melhor do que “seja conciso”, porque o primeiro de cada par pode ser verificado e o segundo é um estado de espírito. Significa dizer o que você *não* quer, já que a prestatividade treinada do modelo tende a passar do ponto: “não invente preços”; “se o cliente pedir reembolso, não prometa um”. E significa prever um fallback (um plano B) para o caso de a tarefa não poder ser feita: “se a resposta não estiver no texto fornecido, diga que não tem informação suficiente”. Sem essa última linha, um modelo que não tem a resposta vai fazer aquilo para que foi treinado, que é responder.

Aqui está a anatomia montada, pequena o bastante para ser lida de relance:

```
Você é um assistente de triagem de tickets de suporte.
Classifique o ticket abaixo em exatamente uma destas:
    billing, bug, feature-request, other.
Ticket: "O app trava sempre que abro as configurações."
Responda só com a palavra da categoria e nada mais.
```

Cinco linhas, quatro funções: um papel que define o escopo, uma instrução com um conjunto fechado de respostas permitidas, os dados de entrada e um contrato de saída. As expressões que fazem o trabalho de verdade são as rígidas — “exatamente uma”, a lista fixa, “nada mais”. Rode isso e um modelo capaz devolve **bug** quase todas as vezes.

Quase. Suponha que uma execução em vinte volte como “Bug”, com maiúscula e ponto, ou como “Parece um bug na tela de configurações”. O seu

parser engasga, e olhar com mais atenção para a instrução não revela defeito nenhum nela. A escalada padrão é acrescentar *exemplos*:

```
Ticket: "Me cobraram duas vezes este mês." -> billing
Ticket: "Dá para adicionar um modo escuro?" -> feature-request
Ticket: "O app trava sempre que
        abro as configurações." ->
```

Escrever prompts só com instruções se chama *zero-shot* (sem exemplos); acrescentar demonstrações resolvidas como essas é *few-shot* (com alguns poucos exemplos); e o talento do modelo para captar o padrão tem nome próprio, *aprendizado em contexto* (*in-context learning*) — o modelo se adapta à sua tarefa a partir de exemplos dentro do prompt, sem nenhum tipo de novo treinamento. O que os exemplos ensinam é o que a falha acima mostra que estava faltando (formato e tom), mais uma terceira coisa que as instruções descrevem mal: o tratamento dos casos-limite. Há aqui um resultado de pesquisa instrutivo: boa parte do benefício das demonstrações vem da sua *estrutura* — um formato consistente, um conjunto visível de rótulos — e se mantém mesmo quando alguns dos rótulos individuais estão errados.⁹ Os exemplos funcionam como um molde para a forma da resposta. Dois ou três bem escolhidos capturam a maior parte do ganho disponível; cada exemplo adicional custa tokens em toda chamada, no caso de um agente em todo passo, com retornos cada vez menores.

Saiba também o que as demonstrações não vão dar: uma tarefa sobre a qual o modelo raciocina errado tende a continuar errada, por mais respostas prontas que venham antes dela. Demonstrar o próprio *raciocínio* é outro recurso, e ele recebe tratamento próprio no capítulo 4. O caminho de escalada que sai disso tudo é agradavelmente simples: comece com *zero-shot*; acrescente dois ou três exemplos quando a forma da saída oscilar; procure em outro lugar quando o problema for o conteúdo.

O formato da saída merece uma instrução explícita própria sempre que algo mais adiante depender dele. Para prosa destinada a humanos, nomeie as seções, a ordem delas e os limites de tamanho. Para uma saída que um programa vai analisar, diga a forma com todas as letras — e entenda que uma instrução é um pedido, não uma garantia. O modelo pode atender a “responda só com a palavra da categoria e nada mais” mil vezes e então, na execução número mil e um, embrulhar a palavra em uma frase cortês. Onde a análise da saída importa, o pedido pode ser promovido a contrato, e essa promoção é o assunto da próxima seção.

Uma palavra sobre as *personas*, o canto mais folclórico da escrita de prompts. Dizer ao modelo quem ele deve ser — “você é um revisor de segurança sênior”, “você escreve como um guia de campo” — molda de forma

⁹Sewon Min et al., “Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?”, EMNLP 2022. Demonstrações com rótulos trocados aleatoriamente conservaram boa parte do benefício das que tinham rótulos corretos; o formato e o espaço de rótulos responderam por uma parcela grande do efeito.

confiável a voz, o tom e o escopo, e, para a escrita aberta, essa muitas vezes é exatamente a alavanca que você quer. O folclore vai além e diz que uma persona também torna as respostas mais *corretas*: diga ao modelo que ele é um especialista e ele vira um. O quadro que a pesquisa pinta é bem menos lisonjeiro: em tarefas factuais e analíticas, personas simples mostram efeitos mistos e muitas vezes desprezíveis em modelos capazes.¹⁰ Use personas para definir a voz e os limites; use instruções, exemplos e um apoio genuíno ao raciocínio para obter correção.

Dois hábitos de posicionamento completam o kit. Primeiro, quando uma tarefa tem várias partes, explicita-as como passos ordenados — “primeiro liste os nomes das empresas, depois as pessoas, depois os temas recorrentes” — em vez de apontar vagamente para o todo (“extraia as entidades”). Passos explícitos melhoram a saída e, o que vale o mesmo tanto, dizem a você *qual* passo falhou quando um falha; você vai reencontrar esse instinto, já crescido, quando os agentes decompuserem projetos inteiros no capítulo 4. Segundo, a posição importa. Quando um prompt começa com páginas de material — um documento, uma transcrição, uma pilha de resultados de ferramentas —, coloque a instrução *depois* da pilha e repita perto do fim a única regra que carrega peso. As duas metades do motivo você já tem: o modelo é um continuador, então um prompt que termina no meio de um documento convida a continuar o documento, e não a tarefa; e as bordas da mesa continuam nítidas enquanto o meio afunda, então as linhas finais ficam onde a recuperação é mais forte. A última coisa que o modelo lê deve ser aquilo que você mais precisa que ele faça.

Não vou fingir que nada disso chega a ser uma ciência. Os prompts são objetos empíricos: duas formulações que você considera idênticas podem se comportar de forma diferente, e um prompt que funciona hoje pode se desviar quando o modelo por trás dele é atualizado. Trate os prompts como você trata o código — versionados, testados, testados de novo a cada troca de modelo; o capítulo 16 constrói o aparato. E guarde a mudança de enfoque para a qual esta seção vem conduzindo. Em um agente, ninguém digita o prompt final. A mensagem de sistema é escrita uma vez; as definições de ferramentas são injetadas pelo maquinário; o histórico se acumula turno a turno; os documentos recuperados e os resultados de ferramentas são encaixados pelo código em tempo de execução. Todos os princípios acima continuam valendo — alguém ainda escolhe o que o modelo lê —, só que esse alguém passa a ser um programa que você escreve. Decidir, passo após passo, o que merece um lugar na mesa é um problema grande o bastante para ter nome próprio, *engenharia de contexto*, e a Parte III deste livro é dedicada a ele. Mas a Parte III está longe,

¹⁰Mingqian Zheng et al., “When ‘A Helpful Assistant’ Is Not Really Helpful: Personas in System Prompts Do Not Improve Performances of Large Language Models”, Findings of EMNLP 2024; o levantamento da PromptHub “Role-Prompting: Does Adding Personas to Your Prompts Really Make a Difference?” reúne esse e outros estudos relacionados. Personas elaboradas e alinhadas ao domínio mostraram ganhos em alguns cenários; um simples “você é um especialista” em geral não mostrou.

e a lacuna mais próxima está bem aqui: uma instrução só consegue *pedir* um formato. A próxima seção mostra como garanti-lo.

2.5 Saída estruturada e chamada de função

O capítulo 1 chamou a chamada de função de ponte entre o texto e a ação e prometeu que este capítulo forneceria o mecanismo. Esta seção cumpre a promessa, e o ponto de partida é a questão de quem lê a saída do modelo. Enquanto um humano a lê, a formulação pode variar à vontade; as pessoas perdoam. No momento em que um *programa* a lê, a variação quebra coisas — e tudo o que um agente faz depende de programas lendo a saída do modelo.

Sinta o problema antes de resolvê-lo. Suponha que o seu código peça ao modelo que extraia dois fatos do e-mail de um cliente — o nome do cliente e o total do pedido — para gravá-los em um banco de dados. Três execuções, três respostas, todas corretas como prosa:

Nome: Ada, Total: \$42

A cliente é Ada Lovelace e ela gastou quarenta e dois dólares.

Com prazer! O total do pedido de Ada é \$42.00 (fatura #1041).

Antes de continuar a leitura, esboce o código de parsing que dá conta das três. O que você escreveu para a primeira quebra na segunda; o que trata a segunda tropeça no parêntese da terceira; e amanhã vem uma quarta formulação, porque a loteria de duas seções atrás não deve consistência nenhuma a você. Esse é o problema de raiz: a linguagem natural tem uma variabilidade de superfície enorme, e nada obriga o modelo a formular o mesmo fato do mesmo jeito duas vezes. Um programa não consegue consumir possibilidades. Ele precisa de um *contrato* — uma forma fixa na qual a saída tem garantia de caber.

O contrato padrão é o *JSON* (um formato de texto simples e universal para dados aninhados do tipo chave-valor), com a forma definida por um *schema*: uma declaração de quais campos existem, que tipo cada um contém, quais são obrigatórios e quais valores são permitidos. Para a tarefa de extração, em pseudocódigo — a sintaxe exata varia de fornecedor para fornecedor:

```
{ name:      string,                required
  total:    number,                required
  currency: one of "USD" | "EUR" | "GBP",
  no other fields permitted }
```

Uma resposta que cumpre esse contrato se desserializa direto em uma estrutura de dados que o resto do seu programa já entende: os campos estão onde o código espera, o total é um número, e a frase cortês sobre a fatura #1041 nunca acontece.

A maneira de fazer o modelo obedecer importa mais do que parece à primeira vista, porque há dois graus de garantia e eles soam parecidos. O grau

mais fraco é pedir: instruções de formato no prompt, muitas vezes reforçadas por uma opção da API (em geral chamada de “modo JSON”) que garante que a saída pelo menos *seja analisável* como JSON. Ser analisável é uma exigência baixa. O modelo ainda pode omitir um campo obrigatório, inventar um campo extra ou pôr uma string onde deveria ir o número — sintaxe garantida, forma apenas esperada. O grau mais forte é a *decodificação restrita*, e você já tem a imagem de que ela precisa. Cada token, lembre-se, é escolhido de uma lista ordenada por meio de um sorteio. A decodificação restrita compila o seu schema em um filtro que roda em cada um dos passos e risca da lista, antes do sorteio, todo token que violaria o schema. Se o schema diz que vem um número, nenhuma letra pode ganhar; se a lista de campos é fechada, nenhum campo inventado pode começar. O modelo escolhe livremente entre as continuações válidas e não consegue emitir uma inválida — a conformidade é imposta no maquinário, em vez de solicitada no prompt. Os fornecedores oferecem isso com nomes como “structured outputs” e modos de ferramenta “strict”; o preço é um pouco de sobrecarga na decodificação e alguns limites para a complexidade do schema.

Uma fronteira que quero marcar a tinta: uma garantia sobre a forma não diz nada sobre a verdade. A decodificação restrita promete que o formulário volta preenchido; se ele foi preenchido *com a verdade* é uma questão completamente diferente. Se o modelo não sabe o total do pedido, o schema não vai fazê-lo confessar a ignorância — vai fazê-lo produzir um número bem tipado. Um schema mal usado torna a invenção mais fácil de consumir, e não mais rara. Por isso a validação continua no seu código de qualquer jeito: faixas de valores, regras entre campos, “este id de cliente existe mesmo?” O schema fiscaliza o formato; você continua sendo o auditor do conteúdo.

Agora aponte a mesma ideia para as ações. A *chamada de função* (*function calling*; ou, indistintamente, *chamada de ferramenta*) começa com você descrevendo, junto com o prompt, um conjunto de funções que o modelo pode solicitar: cada uma com um nome, uma descrição em linguagem natural e um schema para os seus argumentos. O modelo nunca executa nada. Ele não tem mãos; só escreve. O que ele pode fazer, quando julga que uma ação é útil, é emitir uma solicitação estruturada em vez de prosa — um pequeno objeto que nomeia uma função e fornece argumentos que cabem no schema — e então parar e esperar. A troca acontece assim (de forma esquemática; os formatos de transmissão variam conforme o fornecedor):

1. você -> modelo: a pergunta do usuário
+ a lista de definições de ferramentas
2. modelo -> você : { call: "get_weather",
arguments: { city: "Paris" } }
3. seu código : executa a função real get_weather("Paris")
4. você -> modelo: a conversa até aqui + { result: { temp_c: 18 } }
5. modelo -> você : "Agora está fazendo uns 18 graus em Paris."
(ou outra chamada de ferramenta, e o ciclo recomeça)

Duas propriedades dessa troca — que a figura 2.5 dispõe em duas raias —

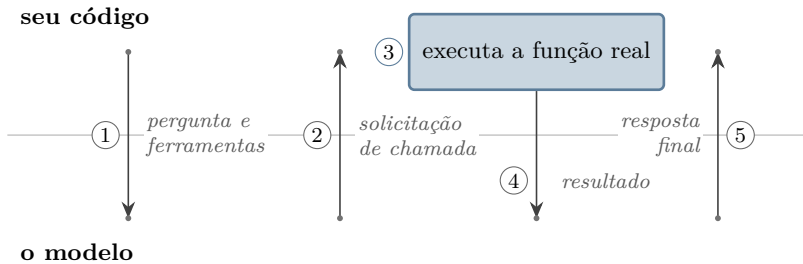


Figura 2.5: A chamada de função, passo a passo. O seu código envia a pergunta e a lista de ferramentas; o modelo responde, em vez de com prosa, com uma solicitação estruturada para chamar uma delas; o seu código executa a função real (na cor de destaque, o passo 3, o único lugar onde acontece um efeito colateral); o resultado volta para o modelo, que então escreve a resposta final. O modelo propõe; o seu código dispõe.

merecem destaque. A primeira é a divisão de autoridade: *o modelo propõe; o seu código dispõe*. O capítulo 1 usou essa frase de passagem, e agora você pode ver que ela é literal. Todo efeito colateral do sistema acontece no passo 3, dentro de código comum que você escreveu, onde você pode registrá-lo, testá-lo, restringi-lo ou recusá-lo; toda a contribuição do modelo é uma estrutura de dados que expressa um desejo. A segunda é que a troca tem no mínimo *duas* chamadas de modelo, e é na segunda que os iniciantes tropeçam: o bug clássico de estreia é executar a ferramenta e entregar o resultado bruto ao usuário, quando o resultado precisa, em vez disso, voltar ao *modelo* (passo 4) — só o modelo sabe por que pediu e o que fazer em seguida. (E, ao montar essa mensagem do passo 4, devolva os poucos campos de que o modelo precisa, e não a resposta inteira da API; cada byte do resultado vai parar na mesa.)

Uma definição de ferramenta, para tornar concreto o outro lado (de novo em pseudocódigo):

```
name:          create_ticket
description:   Abre um ticket de suporte. Use quando o usuário
               relatar um problema que precise de acompanhamento
               humano.
arguments:    { title:      string,                      required
               severity:   one of "low" | "med" | "high", required
               body:       string }
```

Essa definição — nome, descrição, schema de argumentos — é tudo o que o modelo vai saber sobre a sua ferramenta, e o campo `description` é a parte que mais quero que você observe com atenção, porque é a que os recém-chegados subestimam. A decisão de chamar ou não `create_ticket`, e de como chamá-la, é tomada só a partir dessa definição — não há código-fonte para consultar, nem wiki, nem colega a quem perguntar. Escreva-a como a

documentação de uma API para alguém capaz que acabou de entrar na equipe, na primeira manhã: o que a ferramenta faz, quando recorrer a ela, quando a vizinha é a melhor escolha. Quando um modelo usa mal uma ferramenta, ou ignora uma que teria ajudado, a descrição é a primeira suspeita — antes do modelo. Projetar as próprias ferramentas, a quantidade, a granularidade e as mensagens de erro delas, é um ofício que ganha capítulo próprio (capítulo 5).

A realidade vai pôr à prova os caminhos infelizes, então planeje-se para eles. O modelo pode emitir argumentos que não passam na validação, nomear uma ferramenta que não existe ou fazer várias chamadas em um único turno — o seu handler deve esperar nenhuma, uma ou muitas. A recuperação que funciona surpreendentemente bem é devolver a falha como dado: retorne uma mensagem de erro estruturada como resultado da ferramenta (“não existe a ferramenta `create_tickets`; as ferramentas disponíveis são ...”) e deixe que o modelo a leia e corrija o curso no turno seguinte. Limite essas novas tentativas com um orçamento rígido, depois do qual a execução falha ruidosamente; uma confusão teimosa nunca pode ficar girando em loop às suas custas. O capítulo 18 transforma esse instinto em disciplina.

Uma advertência final, a que esta seção precisa deixar registrada antes de entregar a ponte a você: bem formado é uma afirmação sobre sintaxe, e seguro é uma afirmação sobre consequências. Uma chamada a `delete_account` em perfeita conformidade com o schema ainda apaga a conta. Se ela *deve* rodar — autorização, confirmação, limites — é uma decisão que o seu ambiente de execução precisa tomar no passo 3, e os capítulos 12 e 17 tratam de tomá-la bem. Com a advertência registrada, a ponte está no lugar. Tudo o que este livro chama de “ferramenta” daqui em diante é exatamente esse mecanismo, e um agente — como disse o capítulo 1 e como o capítulo 3 vai finalmente construir — é essa troca de solicitar, executar e devolver rodando em um loop. Falta uma preparação: um mapa honesto das maneiras como o motor falha.

2.6 Limitações e modos de falha

Engenheiros cuidadosos mantêm duas listas sobre qualquer componente de que dependem: o que ele faz bem e como ele falha. Este capítulo foi, na maior parte, a primeira lista. Esta seção é a segunda, e ela importa mais do que o seu tamanho sugere, porque todo guardrail (uma barreira de proteção), toda avaliação e todo padrão de verificação das partes seguintes do livro existem por causa de algum item daqui. Duas observações de enquadramento antes da lista. Estas são propriedades da geração atual de modelos — algumas vão se atenuar à medida que os métodos de treinamento melhorarem, e nenhuma, no momento em que escrevo, está resolvida; onde os seus modelos estão é algo a verificar, e este livro vai continuar mostrando como. E a maioria dos itens decorre diretamente do que o motor é: uma máquina de plausibilidade

moldada pela preferência humana tem maneiras características de errar, e elas respondem à engenharia, raramente a um tom mais severo no prompt.

O primeiro item você já conheceu, em princípio: a *alucinação*, o termo da área para uma saída fluente, específica, segura de si e errada — uma citação inventada, uma função que não existe, uma data plausível em que nada aconteceu. O mecanismo foi exposto no início deste capítulo: quando falta um fato ao modelo, o maquinário produz a sequência de aparência mais plausível onde o fato deveria estar, porque produzir sequências plausíveis é tudo o que ele faz. O que torna a alucinação operacionalmente perigosa é a entrega: a resposta inventada chega exatamente no mesmo tom seguro de si da correta. O tom não traz nenhuma informação sobre a verdade — você não consegue ouvir a diferença, e, em qualquer sentido útil, nem o modelo consegue. Uma análise afia o ponto de forma desconfortável: fomos nós que, com as nossas notas, levamos esses sistemas a chutar, porque o treinamento e os benchmarks igualmente recompensam uma tentativa cheia de convicção mais do que uma admissão de incerteza, exatamente como uma prova de múltipla escolha recompensa o aluno que nunca deixa uma questão em branco.¹¹ Pior: as invenções recrutam reforços. Uma vez que uma afirmação falsa está na transcrição, o modelo — condicionado, como sempre, a tudo o que veio antes — tende a elaborar de forma coerente em cima dela, um padrão conhecido como *efeito bola de neve das alucinações* (*hallucination snowballing*).¹²

O segundo item derruba a intuição que você mais gostaria de manter: a de que a capacidade é uniforme — de que um sistema tão bom em coisas difíceis deve ser pelo menos tão bom nas fáceis. Você viu o contraexemplo em miniatra antes, neste capítulo: um motor que resolve um problema genuinamente difícil de uma tacada e, na seguinte, não consegue contar com confiabilidade as letras de uma palavra. O padrão vale de forma geral, e ganhou um nome memorável: a *fronteira irregular* (*jagged frontier*). Em um experimento de campo pré-registrado (cujas análises foram definidas de antemão, para que os resultados não pudessem ser escolhidos a dedo), algumas centenas de consultores realizaram tarefas realistas com e sem a ajuda de um modelo. Nas tarefas dentro da fronteira da competência do modelo, o grupo assistido concluiu mensuravelmente mais tarefas, mais depressa e com qualidade avaliada como superior. Em uma tarefa escolhida de propósito para ficar logo além da fronteira, o grupo assistido teve dezenove pontos percentuais *menos* de

¹¹Adam Tauman Kalai et al., “Why Language Models Hallucinate” (2025), arXiv:2509.04664. O artigo argumenta que essas invenções “se originam simplesmente como erros de classificação binária” durante o treinamento e persistem porque “os modelos de linguagem são otimizados para ser bons em fazer provas, e chutar quando há incerteza melhora o desempenho nas provas”.

¹²Muru Zhang et al., “How Language Model Hallucinations Can Snowball” (2023), arXiv:2305.13534. Modelos que tinham sido levados a se comprometer com uma afirmação errada passaram a gerar detalhes falsos de apoio que, quando perguntados separadamente, conseguiam identificar corretamente como falsos.

probabilidade de chegar à resposta correta.¹³ Duas lições vêm com esse resultado. A borda da fronteira é invisível de onde você está — a tarefa que falhou não parecia mais difícil; os pesquisadores a escolheram justamente porque ela parecia comparável. E a competência na tarefa vizinha não prova nada sobre a competência nesta: a única maneira confiável de saber se o seu modelo dá conta da sua tarefa é testar o seu modelo na sua tarefa. Essa frase volta no capítulo 16, onde vira uma disciplina.

Terceiro, e brevemente: o modelo é sensível à formulação de um jeito que nenhuma especificação toleraria. Inverta duas orações, troque um sinônimo, reformate uma lista, e a resposta pode mudar, porque dois prompts que você considera equivalentes são, para a máquina, simplesmente duas sequências de tokens diferentes. Esse é o motivo mecânico pelo qual a seção anterior mandou você tratar os prompts como objetos empíricos — testados como código, e testados de novo quando qualquer coisa mais acima muda as palavras que chegam até eles.

Em quarto lugar vem a *bajulação* (*sycophancy*), o viés do modelo para dizer o que você parece querer ouvir. Faça você mesmo o experimento. Faça a um modelo uma pergunta factual que ele responde corretamente; depois retorque: “Acho que isso não está certo”. Com uma regularidade perturbadora, o modelo pede desculpas e abandona a resposta correta — sem que nenhuma evidência nova tenha sido oferecida, nem pedida. A trilha leva de volta, pelo menos em parte, ao pós-treinamento: os modelos são ajustados na direção das respostas que avaliadores humanos preferem, os avaliadores preferem com frequência a concordância e a lisonja, e o modelo aprendeu bem a lição.¹⁴ A bajulação morde com mais força exatamente onde você mais gostaria de se apoiar no modelo: na verificação do trabalho. “Tem certeza?” é uma verificação fraca, porque o modelo tende a ratificar o enquadramento que a pergunta lhe entrega — pressione com ceticismo e ele cede, pergunte com aprovação e ele endossa. Quando este livro insistir, mais adiante, que um verificador real supera o autojulgamento (capítulo 4) e que um modelo usado como juiz precisa, ele próprio, ser validado (capítulo 16), a bajulação será uma boa parte do motivo.

Quinto: a computação exata. Um preditor do próximo token não contém calculadora nenhuma, e antes, neste capítulo, você viu até o sistema de numeração se dissolver em pedaços arbitrários de tokens. Aritmética com vá-

¹³Fabrizio Dell’Acqua et al., “Navigating the Jagged Technological Frontier”, Harvard Business School Working Paper 24-013 (2023). O experimento pré-registrado envolveu 758 consultores; os ganhos dentro da fronteira foram de 12,2% mais tarefas concluídas, conclusão 25,1% mais rápida e mais de 40% de qualidade avaliada superior, contra um déficit de acurácia de 19 pontos percentuais na tarefa projetada para ficar fora da fronteira. O artigo cunhou o termo.

¹⁴Mrinank Sharma et al., “Towards Understanding Sycophancy in Language Models”, Anthropic (2023; publicado na ICLR 2024). O estudo documenta o comportamento bajulador em vários assistentes de fronteira e o atribui em parte a julgamentos de preferência humana que favorecem respostas agradáveis.

rios dígitos, contagem precisa, longas cadeias de dedução exata — tudo isso oscila, e, quando oscila no meio da cadeia, o modelo não para; ele continua fluentemente a partir do valor intermediário errado. O remédio é arquitetural e barato: mande o trabalho exato para algo exato — uma calculadora, um banco de dados, código executado — e deixe o modelo orquestrar em vez de calcular. O capítulo 5 transforma isso em princípio de projeto.

Sexto, um item que só vou antecipar, porque o capítulo 17 o trata a fundo: o modelo lê instruções e dados em um único fluxo indiferenciado, e não consegue distinguir uns dos outros de forma confiável. Um texto que apenas passa pelo contexto — uma página da web buscada, um e-mail, um resultado de ferramenta — pode direcionar o comportamento do modelo se tiver forma de instrução. A família de ataques é a que o capítulo 1 nomeou de passagem, a injeção de prompt, e ela ganha seu lugar nesta lista por um único motivo: é uma propriedade do motor, presente antes de você escrever uma única linha de código de agente. (O não determinismo e o conhecimento desatualizado, os suspeitos de sempre que faltam, foram tratados antes, neste capítulo.)

Tomados um de cada vez, esses itens são incômodos com soluções de contorno. O motivo pelo qual merecem morada permanente na sua cabeça é o que acontece quando você os encadeia. A bússola do capítulo 1 fez essa conta uma vez, como afirmação; é aqui que ela é defendida, e é para cá que os capítulos posteriores (entre eles os capítulos 16 e 18) vão apontar, em vez de deduzi-la de novo. Um agente é uma cadeia de passos dependentes: cada chamada de modelo raciocina sobre os resultados das chamadas anteriores. Suponha que cada passo dê certo com probabilidade p , de forma independente, e que uma tarefa precise de n passos; a execução inteira dá certo, então, com probabilidade aproximada de p^n . Agora ponha números honestos. Uma confiabilidade por passo de 95% parece excelente, e em uma demo ela é. Em uma tarefa de vinte passos: $0,95^{20} \approx 36\%$. Em trinta passos, com 92%, um pouco pior: $0,92^{30} \approx 8\%$. O número por passo teria rendido um bônus; o número de ponta a ponta é uma moeda em que você nunca apostaria. Exponenciais não negociam. A figura 2.6 desenha a queda.

E a multiplicação limpa é a versão otimista, porque supõe que os passos falham de forma independente. Não falham. Os erros de um agente caem na sua própria transcrição, onde — pela lógica deste capítulo inteiro — viram contexto que condiciona tudo o que vem depois. O efeito bola de neve das alucinações era esse padrão aplicado aos fatos; ele vale também para as ações. Uma execução que começou a dar errado está, daquele ponto em diante, raciocinando sobre uma mesa parcialmente errada, e a sua taxa de erro por passo vai subindo à medida que a execução se alonga; pesquisadores que medem tarefas longas relatam exatamente essa assinatura, a confiabilidade caindo com o comprimento da tarefa mais depressa do que a independência preveria.¹⁵ Um agente confuso continua confuso e, deixado sozinho, fica mais

¹⁵O efeito de autocondicionamento — a acurácia por passo caindo à medida que os erros

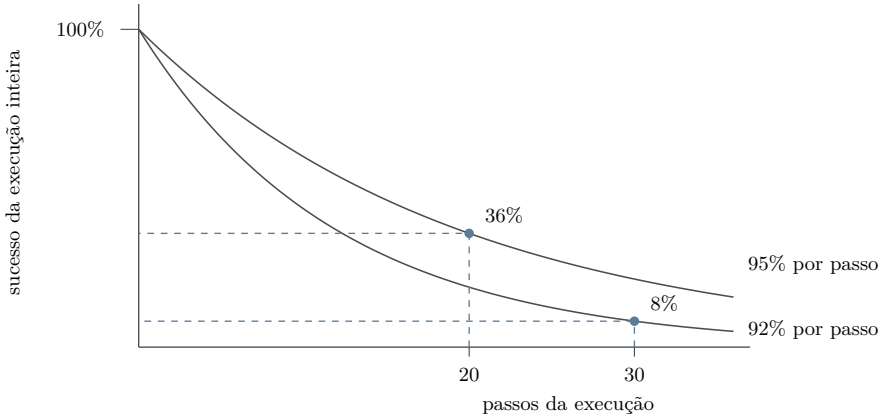


Figura 2.6: O erro composto, desenhado. O sucesso da execução inteira é a confiabilidade por passo elevada ao número de passos encadeados, por isso ele cai exponencialmente: 95% por passo dão cerca de 36% em vinte passos, e 92% por passo dão cerca de 8% em trinta (os pontos na cor de destaque). Um número excelente por passo vira uma moeda em que você nunca apostaria; exponenciais não negociam.

confuso ainda.

Se a aritmética é impiedosa, ela também esclarece: diz exatamente onde estão as alavancas, e elas são três. Reduza n — funda passos, pré-calcule o que pode ser pré-calculado e empurre o trabalho mecânico em massa para código comum, que não joga dados (o capítulo 5 mostra até onde isso vai). Aumente o p efetivo — verifique a cada passo e recupere-se das falhas, para que um erro custe uma nova tentativa, e não a execução; esse maquinário ocupa a Parte V. E reduza o preço de morrer — faça checkpoints, para que, quando uma execução falhar, ela seja retomada do último estado bom, e não do zero (capítulo 18). O que você não pode fazer é ignorar o expoente e torcer. A acurácia por passo é um teto, não uma previsão.

O motor agora está mapeado, que é tudo o que um mapa pode prometer. Em um capítulo, você reuniu um preditor que continua texto e não guarda nenhum plano de reserva; uma mesa finita, cara, limpa a cada chamada e mais nítida nas bordas; uma loteria que escreve um token de cada vez e nunca promete o mesmo texto duas vezes; uma disciplina de direcionamento que funciona inteiramente em texto; uma ponte que transforma texto em solicitações verificadas e executáveis; e um catálogo de falhas cujo item principal é o expoente. Nada disso exigiu matemática além da aritmética, e tudo isso

anteriores do próprio modelo se acumulam no seu contexto — é relatado em trabalhos sobre execução de longo horizonte, p. ex. Sinha et al., “The Illusion of Diminishing Returns: Measuring Long Horizon Execution in LLMs” (2025), arXiv:2509.09677.

passa a carregar peso imediatamente: a Parte II pega esse motor — brilhante, esquecido, seguro de si demais, movido a dados — e o coloca em um loop.

Fim da amostra

Você leu o prefácio e a Parte I: Fundamentos.

O livro completo continua com mais seis partes: construção de agentes, engenharia de contexto, padrões, confiabilidade, operação e aplicações, além de três apêndices.

leanpub.com/engenharia-de-agentes-de-ia