J.C. KÖDEL

CONTEXT ENGINEERING

ENGENHARIA DE INFORMAÇÃO PARA SISTEMAS DE IA

ESPECIFICAÇÕES | ARQUITETURA | CÓDIGO | ADRs | TESTES | DOCUMENTAÇÃO | REPOSITÓRIO

Sumário

Esta é uma amostra grátis. A edição completa traz todos os capítulos.

- Loops de desenvolvimento com IA *(só na edição completa)*
- Medindo contexto: como avaliar se seu contexto melhora resultados *(só na edição completa)*
- Princípios aplicados: chat, IDE, terminal e CI *(só na edição completa)*
- Times: contexto como ativo do repositório *(só na edição completa)*
- Preparando um projeto (do zero e um legado) *(só na edição completa)*
- Implementação completa guiada por IA *(só na edição completa)*
- Post-mortem: onde o contexto falhou e como foi recuperado *(só na edição completa)*
- Referências *(só na edição completa)*

Sobre o autor

Comecei a programar nos anos 90, escrevendo software para locadoras de vídeo. Em 1998 construí meu primeiro ERP em Visual Basic 6 com SQL Server. Clientes reais usaram o sistema por anos, e ele degradou nas minhas mãos por falta de método. Essa foi a primeira lição cara da carreira: o difícil raramente é escrever código, é sustentar o que você escreveu.

A partir de 2002 passei a trabalhar em sistemas sem margem para falha: cadastros internacionais e controle de acesso na Polícia Federal, internet banking internacional e a implementação de Basileia II. Um framework que escrevi nessa época segue em produção num grande banco quase vinte anos depois. Mais adiante, trabalhei num sistema de inteligência artificial que analisava 10 milhões de ligações por mês, numa operação de atendimento com mais de 150 mil funcionários em 13 países.

Em 2017 lancei um produto meu, o Meu Cronograma Capilar. O app passou de 10 milhões de downloads, mantém nota 4,8 e ocupa o Top 10 da categoria na Play Store desde 2018. Cuido de tudo sozinho: código, arquitetura, testes, operação e publicação. Cito o app porque ele prova algo que nenhum cargo prova: ciclo completo, entregue e sustentado por quase uma década, sem equipe para compensar atalho.

Hoje construo software com IA em produção todos os dias. E o que a prática diária me mostrou é que a diferença entre um resultado consistente e um chute caro quase nunca está no modelo: está na informação que você entrega a ele. Essa percepção virou este livro, que fecha uma trilogia. *Spec Driven Development* (2026, https://books.kodel.com.br/pt-br/livros/sdd) ensina o quê construir, trocando prompts soltos por especificações. *FOCUS Architecture* (2026, https://books.kodel.com.br/pt-br/livros/focus) ensina onde a regra de negócio mora, para que humanos e IA saibam onde mexer. Este ensina como alimentar a IA com a informação certa, no momento certo. Nenhum dos dois é pré-requisito: você pode ler este volume sem ter aberto os outros. E este livro foi produzido com as próprias técnicas que ensina, do rascunho à revisão; na minha opinião, é o mínimo que você deveria exigir de quem escreve sobre o assunto.

O que afirmo nas próximas páginas, vi funcionar em produção ou digo de quem aprendi. Nada aqui é teoria de palco.

J.C.Ködel

Como LLMs utilizam contexto

Na terça-feira você pediu ao assistente de IA uma função de importação de planilha e recebeu código limpo, no padrão do seu projeto, com os casos de erro tratados. Na quinta você pediu praticamente a mesma coisa e recebeu um script solto, com nomes genéricos, ignorando as convenções que o próprio assistente tinha seguido dois dias antes. O pedido era o mesmo. O modelo era o mesmo. O resultado, oposto.

A explicação mais comum que você vai ouvir é alguma variação de "IA é assim mesmo, é loteria". Essa explicação é cômoda e errada. Existe uma variável concreta que mudou entre a terça e a quinta, e ela tem nome: o contexto. Na terça, a conversa já continha trechos do seu código, a discussão sobre o padrão de erros e dois exemplos seus. Na quinta, você abriu uma sessão nova e mandou o pedido a seco. O modelo não piorou. A informação que ele recebeu piorou.

Este capítulo estabelece o modelo mental que sustenta o livro inteiro: o que exatamente o modelo enxerga quando responde, e o que ele não enxerga de jeito nenhum. Sem isso, toda técnica das próximas partes vira receita decorada.

O modelo só enxerga a entrada

Comece pelo termo que dá nome ao livro. Contexto é tudo que o modelo recebe como entrada em uma chamada: sua pergunta, o histórico da conversa, as instruções de sistema, os arquivos colados, os resultados de ferramentas. Tudo, concatenado, forma um único bloco de texto que o modelo lê de uma vez. No exemplo da terça-feira, o contexto era seu pedido mais os trechos de código e a discussão anterior; na quinta, era só o pedido.

Cada vez que o modelo processa um contexto e produz uma resposta, acontece uma inferência. Inferência é uma chamada ao modelo: entra texto, sai texto, e nada mais participa. Não existe um canal lateral por onde o modelo consulta seu repositório, suas intenções ou a conversa de ontem. Se a informação não está na entrada daquela chamada, para o modelo ela não existe.

Isso soa óbvio escrito assim, mas quase ninguém opera como se fosse verdade. Quando você reclama que "a IA deveria saber" que seu projeto usa determinado padrão, você está atribuindo ao modelo um conhecimento que nunca entrou pela única porta que existe: a entrada. O modelo não deveria saber. Você deveria ter contado.

A consequência prática inverte a pergunta habitual. Em vez de "por que o modelo errou?", pergunte "o que havia na entrada que justificasse a resposta certa?". Na maioria das sessões frustrantes que você já teve, a resposta verdadeira é: nada. O pedido a seco da quinta-feira não continha o padrão de erros do projeto, então o modelo escolheu um padrão qualquer. Do ponto de vista dele, a resposta da quinta foi tão boa quanto a da terça: coerente com a entrada recebida.

Atenção: como o modelo pesa o que você mandou

Dentro de uma inferência, o modelo não trata o contexto como uma sacola uniforme de palavras. A arquitetura por trás dos LLMs (**L**arge **L**anguage **M**odels, ou *grandes modelos de linguagem*) atuais, descrita por Vaswani e coautores no artigo "Attention Is All You Need" (*Atenção é tudo que você precisa*, 2017, arXiv:1706.03762), gira em torno de um mecanismo chamado atenção. Para os propósitos deste livro, basta o modelo mental: ao gerar cada palavra da resposta, o modelo atribui pesos a cada trecho da entrada, decidindo o quanto cada pedaço influencia a próxima palavra. Trechos com peso alto puxam a resposta; trechos com peso baixo quase não participam.

Um exemplo mínimo. Suponha a entrada: "Nosso backend é em Go. Responda sempre com exemplos na linguagem do backend. Como abro um arquivo?". Ao gerar a resposta, a atenção liga "linguagem do backend" a "Go", e o exemplo sai em Go. Se a frase sobre o backend estivesse ausente, o peso se distribuiria para o resto e o modelo escolheria a linguagem mais provável dado o restante da conversa, talvez Python. A resposta muda sem que a pergunta final tenha mudado uma letra.

Duas consequências desse mecanismo importam para você. A primeira: tudo que está no contexto participa da disputa por atenção, inclusive o que você colou sem pensar. Aquele log de 300 linhas que você jogou na conversa para ilustrar um erro continua lá, recebendo pesos, competindo com as suas instruções a cada palavra gerada. A segunda: atenção é um mecanismo estatístico, não uma busca exata. O modelo não "encontra" a sua instrução como um grep encontra uma string; ele a pesa contra todo o resto. Instruções podem perder a disputa. Os capítulos 4 e 5 mostram quando e por que isso acontece com frequência crescente.

Se você quiser descer ao mecanismo real, com as matrizes e as cabeças de atenção, o artigo de Vaswani e coautores, de 2017 (arXiv:1706.03762), é a fonte primária. Para usar IA bem, o modelo mental acima é suficiente, e este livro não vai além dele.

Nada sobrevive entre chamadas

Falta a peça que derruba a ilusão mais cara: a de que o modelo lembra.

Um LLM é stateless entre chamadas: ele não guarda estado. Terminada uma inferência, o modelo não retém nada do que processou. A documentação pública das APIs de chat dos grandes provedores (as docs da Messages API da Anthropic e da API da OpenAI, em 2026) descreve o mesmo contrato: cada requisição envia a lista completa de mensagens da conversa, e o servidor responde àquela lista. Não há sessão viva do outro lado, não há um "cérebro" que acompanha você entre uma pergunta e outra. Há uma função: entra contexto, sai resposta, fim.

Se ajudar, pense numa central de atendimento peculiar. Toda vez que você liga para essa empresa, quem atende é uma pessoa completamente diferente, sem nenhum acesso ao que você tratou nas ligações anteriores: não há CRM, não há histórico, não há "conforme conversamos ontem". Tudo que essa atendente sabe do seu caso é o que você disser nesta ligação. Em compensação, ela é a atendente mais preparada do planeta: conhece todas as linguagens, todos os frameworks, todos os padrões já publicados. E é exatamente essa amplitude que cria o problema. Diante de um pedido vago, ela não tem como saber qual das mil respostas corretas que conhece é a certa para o seu caso, então escolhe a mais provável em geral, que raramente é a sua. Todo o conhecimento do mundo, sem foco, produz uma resposta genérica; o foco quem traz é você, no que fala durante a ligação. Cada chamada ao modelo é essa ligação: recomeça do zero, com um interlocutor que sabe tudo e não lembra de nada.

"Mas o chat lembra da conversa", você dirá, "ele responde à minha segunda pergunta sabendo da primeira". Responde porque a interface de chat reenvia a conversa inteira a cada mensagem sua. A memória que você percebe não mora no modelo; mora no texto que a ferramenta acumula e reenvia. É um truque de palco legítimo, e o capítulo 3 o desmonta em detalhe, com um transcript real de onde ele quebra.

Por ora, guarde o contrato: uma chamada, um contexto, uma resposta, nenhum resíduo. Esse contrato explica a quinta-feira do início do capítulo por inteiro. A sessão nova não tinha acesso algum à sessão da terça, porque não existe lugar onde a terça pudesse ter ficado guardada. Você não perdeu qualidade do modelo entre um dia e outro; você perdeu o contexto, e com ele foi a qualidade.

Onde o contexto se esconde

Antes de fechar o modelo mental, vale desfazer uma segunda ilusão: a de que o contexto é só o que você digita. Na prática, o texto que você escreve costuma ser a menor parte do que o modelo recebe.

Quando você usa um assistente de código, a ferramenta monta a entrada por conta própria antes de chamar o modelo. Ela costuma incluir uma instrução de sistema (o texto que define o comportamento do assistente), arquivos de configuração do projeto que você talvez nem lembra que existem, trechos dos arquivos abertos no editor, resultados de buscas que a própria ferramenta executou e a saída de cada comando que rodou. Você digita uma linha; o modelo recebe dezenas de milhares de palavras.

Faça o teste na sua ferramenta: procure a opção que exibe a requisição enviada ou o consumo da sessão. A primeira vez que você vê o pacote inteiro costuma ser desconfortável, como abrir o payload de uma requisição que você achava enxuta e encontrar megabytes de penduricalho. E cada um desses penduricalhos, você agora sabe, compete por atenção com a sua instrução.

Isso não é um defeito das ferramentas; é o trabalho delas. Montar contexto automaticamente é o que torna um assistente de código mais útil que um chat pelado. Mas transfere para você uma responsabilidade nova: saber o que está sendo montado em seu nome. Quem nunca olhou a entrada real não tem como diagnosticar por que a saída veio errada. Ao longo do livro, “olhe o contexto” vai aparecer como o primeiro passo de quase todo diagnóstico, do mesmo jeito que “olhe o log” é o primeiro passo de quase toda investigação de produção.

O que muda na sua prática

Junte as três peças. O modelo só enxerga a entrada. Dentro da entrada, a atenção pesa cada trecho, e tudo compete. Entre chamadas, nada persiste. Dessas três frases sai uma definição de trabalho que o restante do livro só refina: a qualidade da resposta é função da qualidade da informação presente no contexto daquela chamada.

Note o que essa definição faz com a sua margem de ação. Você não controla os pesos do modelo, não controla o treinamento, não controla a arquitetura. Você controla uma única coisa: o que entra no contexto. Essa única coisa determina, mais do que qualquer outra variável ao seu alcance, se você recebe o código da terça ou o script da quinta.

Aqui assumo uma opinião, em primeira pessoa: depois de anos usando IA em produção diariamente, não vi nenhum ajuste de ferramenta, modelo ou fraseado render tanto quanto tratar a entrada com o mesmo cuidado que trato uma interface pública. É essa prática que este livro sistematiza.

Falta, porém, uma dimensão que este capítulo tratou como abstrata: a entrada tem um tamanho, e esse tamanho tem um limite e um preço. O modelo não lê "texto", ele lê tokens, e a janela de contexto que os recebe é um recurso finito. O próximo capítulo define essas duas medidas, porque sem elas você não consegue raciocinar sobre o que cabe, o que custa e o que fica de fora.

Tokens e janelas de contexto

Você pede ao agente uma análise do seu projeto. Ele sai lendo arquivo atrás de arquivo, rodando buscas, despejando saídas de comando, e a sessão avança bem até que, sem aviso, a ferramenta anuncia que vai "compactar a conversa" ou simplesmente começa a responder ignorando instruções que você deu vinte minutos atrás. Ninguém te mostrou o que foi acumulado, quanto cabia, ou quanto cada leitura pesou. Você está negociando com um limite invisível, usando uma unidade que não conhece.

O capítulo anterior estabeleceu que a resposta é função do que está na entrada. Este capítulo dá as duas medidas dessa entrada: a unidade em que ela é contada, o token, e o recipiente que a limita, a janela de contexto. Com as duas você passa a estimar o que cabe, prever quando vai estourar e, nos capítulos 5 e 6, calcular o que a folga desperdiçada custa em qualidade e em dinheiro.

O modelo não lê palavras, lê tokens

Um token é a unidade mínima de texto que o modelo processa: um pedaço de palavra, uma palavra curta inteira, um sinal de pontuação, um espaço. Antes de qualquer inferência, o texto do contexto passa por um tokenizador, que o fatia nessas unidades e as converte em números. O modelo nunca vê letras; vê a sequência de números que o tokenizador produziu.

O algoritmo de fatiamento dominante deriva do BPE (**B**yte **P**air **E**ncoding, ou *codificação por pares de bytes*), descrito para uso em modelos de linguagem por Sennrich, Haddow e Birch em 2016 (arXiv:1508.07909). O princípio cabe em uma frase: sequências de caracteres que aparecem com frequência no corpus de treino viram um token único; sequências raras são quebradas em pedaços menores. "the" em inglês é um token só. Um identificador como `calculateTotalWithDiscount` vira vários.

Um exemplo mínimo, usando o tokenizador de código aberto tiktoken, que a OpenAI publica no GitHub (github.com/openai/tiktoken): a frase em inglês "the quick brown fox" (*a rápida raposa marrom*) ocupa 4 tokens, um por palavra, porque todas são comuns. A frase em português "a raposa marrom veloz" ocupa mais, porque o corpus de treino desses tokenizadores é dominado por inglês e as palavras em português são fatiadas com mais frequência. Dessa assimetria saem as duas regras de bolso que você vai usar todo dia: em

inglês, um token equivale a aproximadamente 4 caracteres, ou cerca de três quartos de uma palavra; em português, espere gastar na casa de 20 a 40% mais tokens para dizer a mesma coisa. São aproximações, e a única medida exata é rodar o tokenizador da sua ferramenta, mas para estimar ordem de grandeza elas bastam.

Traga a régua para o seu dia. Uma página de texto corrido fica na casa de algumas centenas de tokens. Um arquivo de código de 300 linhas, alguns milhares. A saída daquele comando de build que o agente rodou e capturou inteira, dezenas de milhares. E aqui está a diferença da era dos agentes: não é mais você quem cola texto na conversa; é o agente quem o acumula, um `Read` de cada vez, uma busca de cada vez, um log de cada vez, e tudo isso entra na mesma conta. Você não precisa de precisão; precisa parar de tratar essas leituras como algo sem peso. Faça o exercício uma única vez para calibrar o olho: pegue um arquivo que o agente leu na sua última sessão, conte os caracteres, divida por quatro e desconte a diferença do português. O número que sair, compare com o consumo que a ferramenta reporta na sessão. Depois da primeira medição, você nunca mais manda um agente "ler o projeto inteiro" sem um segundo de hesitação, e essa hesitação é exatamente o hábito que este capítulo quer instalar. Cada leitura tem um tamanho em tokens, e a partir do próximo capítulo esse tamanho vira protagonista.

O que a tokenização explica de graça

Saber que o modelo enxerga tokens, e não letras, desfaz alguns mistérios que você provavelmente já presenciou e atribuiu a burrice do modelo.

Antes do exemplo, um lembrete que desarma a frustração: um LLM não é uma entidade inteligente no sentido que a conversa fluente sugere. O mecanismo inteiro é um só, prever o texto mais provável na saída dado o texto na entrada. Não há compreensão, intenção ou um "alguém" do outro lado que saiba o que está dizendo; a inteligência que você percebe é uma atribuição sua, induzida pela fluência. Se essa predição constitui alguma forma de inteligência é um debate em aberto na pesquisa, mas o que importa aqui é o mecanismo: quem espera estar conversando com algo verdadeiramente inteligente cobra do modelo habilidades que o mecanismo simplesmente não tem.

O clássico: você pede para contar quantas letras "r" existem em uma palavra e o modelo erra uma conta que uma criança acerta. O erro só choca por causa da expectativa acima; você assumiu inteligência onde há predição de texto. Frustrante, até você lembrar que o modelo nunca viu as letras. A palavra chegou como um ou dois tokens, números inteiros numa sequência, e a composição interna de cada token em caracteres não faz parte do que o

modelo processa diretamente. Pedir para ele contar letras é como pedir para você contar os bytes de uma imagem olhando para a foto: a informação existe em algum nível da representação, mas não no nível em que você opera.

O mesmo raciocínio explica por que trocar uma palavra por um sinônimo às vezes muda o comportamento da resposta mais do que deveria: palavras diferentes fatiam em tokens diferentes, com vizinhanças estatísticas diferentes no treino. E explica por que identificadores de código longos e cheios de abreviações consomem mais tokens do que nomes limpos: o tokenizador fatia o que nunca viu. Nenhum desses efeitos exige que você decore o vocabulário do tokenizador; exige só que você lembre que existe uma camada de fatiamento entre o seu texto e o modelo, e que essa camada tem um custo contável.

É esse custo contável que interessa daqui em diante. Se cada pedaço de texto tem um preço em tokens, a pergunta seguinte é inevitável: quantos tokens cabem numa chamada?

A janela de contexto é o recipiente

A segunda medida é o limite. A janela de contexto é a quantidade máxima de tokens que uma chamada ao modelo comporta, somando tudo: instruções de sistema, histórico da conversa, cada arquivo que o agente leu, cada saída de comando que capturou, e a resposta que o modelo vai gerar. A resposta também conta, porque o modelo gera token a token dentro da mesma janela em que leu a entrada. E, nos modelos de raciocínio que em julho de 2026 são o padrão dos provedores principais, a resposta que você lê não é tudo que o modelo gerou: antes dela vêm os tokens de raciocínio (thinking tokens, *tokens de pensamento*), o rascunho interno que a ferramenta esconde ou resume, e que ocupa janela como qualquer outro token gerado. Uma resposta de dez linhas pode ter custado alguns milhares de tokens de rascunho, e a contabilidade que ignora essa parcela invisível fecha a janela mais cedo do que a conta previa. Quando o total se aproxima do teto, algo tem que ceder: a chamada falha, a resposta sai truncada, ou a ferramenta compacta ou descarta parte do histórico para abrir espaço, como na cena que abriu este capítulo. Das três, a terceira é a mais traiçoeira, e o capítulo 3 mostra o estrago que ela faz.

Quanto mede a janela? Aqui este livro se recusa a imprimir uma tabela, de propósito. Números de janela envelhecem em meses, e um livro que os fixasse estaria mentindo para você antes da segunda impressão. Fique com a ordem de grandeza, ancorada no tempo: em 2026, as janelas dos modelos de ponta dos grandes provedores (Anthropic, OpenAI, Google) ficam entre centenas de milhares e alguns milhões de tokens, e os números exatos estão nas páginas públicas de cada modelo, a um clique. Quando você ler este parágrafo, os valores terão crescido. A mecânica descrita aqui, não.

Faça a conta que interessa: uma janela de centenas de milhares de tokens comporta, grosso modo, algumas centenas de páginas de texto ou um projeto de código pequeno inteiro. Parece muito. E aqui mora a armadilha que separa quem leu este livro de quem leu a página de marketing.

Janela grande não é licença para encher

A reação natural ao crescimento das janelas é: "ótimo, agora mando o agente ler o repositório inteiro e deixo o modelo se virar". Essa reação assume que a janela funciona como um disco, onde ocupar 10% ou 90% dá no mesmo desde que caiba. Não funciona.

Lembre do capítulo 1: dentro da janela, cada token participa da disputa por atenção a cada palavra gerada. Encher a janela muda a disputa. A sua instrução, que numa entrada enxuta dominava os pesos, agora compete com dezenas de milhares de tokens de log, código morto e conversa velha. Existe pesquisa publicada medindo o quanto a qualidade cai conforme o contexto cresce e onde a queda é pior, e o capítulo 5 é inteiro sobre ela. Por ora, registre a assimetria: a janela cresceu porque é um número fácil de vender, mas capacidade de comportar tokens e capacidade de usar bem esses tokens são coisas diferentes, e a segunda não acompanhou a primeira.

Há ainda o lado que nenhuma página de modelo destaca: cada token da janela é cobrado. Os provedores precificam por milhão de tokens de entrada e de saída, então uma janela cheia de lixo custa dinheiro real a cada chamada, mesmo quando a qualidade sobrevive. O capítulo 6 faz essa conta com você, em números.

Meça você mesmo: o que viaja junto com uma pergunta de uma linha

Você não precisa acreditar em números abstratos; o experimento cabe em um prompt. Durante a escrita deste capítulo, abri uma sessão nova do meu agente de código, limpei o histórico e mandei exatamente isto:

> Eu quero que você me retorne exatamente o que você recebeu neste pedido. A intenção aqui é analisar o que está sendo enviado no contexto além do meu prompt.

Duas frases. Umas poucas dezenas de tokens. A resposta enumerou o que mais estava na janela naquela chamada, e a lista é longa: a instrução de sistema da ferramenta, com regras de comportamento e o estado completo do repositório; os schemas de todas as ferramentas que o agente pode chamar, mais uma lista de outra centena de ferramentas disponíveis sob

demanda; o meu arquivo de instruções globais, que arrastava para dentro um guia inteiro de estilo Flutter, inútil para um projeto que não era Flutter; um modo de comportamento injetado por um hook da sessão; as descrições de cerca de trinta e cinco skills instaladas; e instruções de três servidores MCP. Somado, o material que viajou junto com a minha pergunta media dezenas de milhares de tokens, na casa de três ordens de grandeza acima do prompt em si.

Nada disso é defeito da ferramenta; é o preço de um agente equipado. O ponto é outro: cada chamada seguinte da sessão recarrega essa bagagem, e boa parte dela eu mesmo tinha colocado lá e esquecido. Rode o mesmo prompt na sua ferramenta antes de continuar a leitura: saber o que a sua pergunta de uma linha carrega nas costas é o primeiro ato de context engineering que este livro pede a você.

A régua que você leva deste capítulo

Recapitule as medidas com o caso concreto da abertura. Naquela sessão, o agente leu, digamos, uma dúzia de arquivos de algumas centenas de linhas cada, mais duas saídas de build. A alguns milhares de tokens por arquivo e dezenas de milhares por log, a soma passa de cem mil tokens antes de você perceber. Somada à instrução de sistema da ferramenta, ao histórico da conversa e ao espaço necessário para as respostas, o total encostou no teto da janela, e a ferramenta começou a descartar história para sobreviver, levando junto as suas instruções. Nada disso era invisível; era só não medido. Agora você mede: estima os tokens de cada peça, sabe que a soma disputa uma janela finita e sabe que encher a janela tem custo dobrado, em atenção e em dinheiro.

Falta uma peça para o mecanismo fechar. Se cada chamada é isolada, como o capítulo 1 mostrou, e cada chamada carrega no máximo uma janela de tokens, como este capítulo mediu, então de onde vem a sensação de que o chat “lembra” do que você disse dez mensagens atrás? A resposta é um truque de reenvio que as ferramentas fazem por você, e entender esse truque explica por que sessões longas esquecem o combinado. É o próximo capítulo.