EXAME NACIONAL DO ESTOQUE MÉDIO

GABARITO OFICIAL COMENTADO

ATACADÃO 2026

CADERNO 1 • LARANJA

01	02	03	04	05	06	07	08	09	10	11
C	E	A	B	C	D	B	E	A	D	B

Cada questão vem com uma aula curta: primeiro o tema, com fórmulas, diagramas e um exemplo resolvido; depois, as cinco alternativas, uma por uma. No fim, dizemos de onde veio a pergunta: **relato direto** é quando o candidato contou o que caiu; **compilação** é quando um site reúne relatos e atribui a pergunta a uma empresa.

QUESTÃO 01

Gabarito **C**

KV cache: por que a geração não refaz o passado

ENTENDENDO O TEMA

O PROBLEMA

Um modelo de linguagem escreve um *token* por vez. Para escolher o próximo, ele precisa olhar para tudo o que veio antes. No código ingênuo da questão, isso é feito do jeito mais caro: a cada passo o modelo recebe a sequência inteira e refaz a conta do começo. No passo 500, ele processa 1.499 posições para produzir um único *token* novo, e 1.498 delas ele já tinha processado no passo anterior.

A pergunta que leva ao KV *cache* é simples: se o começo do texto não mudou, alguma coisa da conta antiga pode ser guardada e reaproveitada? Para responder, precisamos lembrar como um *token* enxerga os anteriores.

COMO UM TOKEN ENXERGA OS ANTERIORES

Em cada camada, cada *token* vira três vetores, calculados com matrizes de pesos aprendidas: a consulta q (*query*, o que ele procura), a chave k (*key*, a etiqueta que ele mostra aos outros) e o valor v (*value*, o conteúdo que ele entrega). A atenção compara a consulta com as chaves, transforma as notas em pesos e devolve uma média ponderada dos valores:

$$\text{Atenção}(Q,K,V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right)V$$

A matriz M é a máscara causal. Ela vale 0 quando a chave vem antes da consulta (ou é a própria) e $-\infty$ quando vem depois. Depois do *softmax*, $-\infty$ vira peso zero. Em português: **nenhum *token* enxerga o futuro**. É o que permite treinar o modelo para prever a próxima palavra sem colar a resposta.

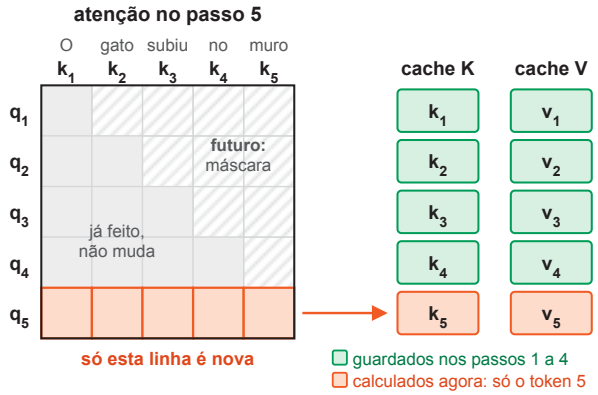


Figura 1. No passo 5, a única consulta nova é a do "muro" (q_5). Ela olha as cinco chaves e mistura os cinco valores. As linhas de cima já foram calculadas nos passos anteriores e a máscara causal garante que não mudam. Por isso basta guardar K e V e calcular só a linha laranja.

POR QUE K E V DO PASSADO NÃO MUDAM

Olhe a Figura 1. Quando o "muro" chega, as linhas de cima da matriz continuam iguais, porque a máscara impede que "O", "gato", "subiu" e "no" olhem para ele. Isso vale camada por camada. Na primeira camada, a saída da posição i depende só dos *tokens* até i. A segunda camada recebe essas saídas e, de novo, a posição i só olha para trás. As outras peças de cada camada (normalização, rede *feed-forward*, soma residual) agem em cada posição separada. Então, em qualquer camada ℓ:

$$k_i^{(\ell)} = h_i^{(\ell-1)} W_K^{(\ell)}, \qquad v_i^{(\ell)} = h_i^{(\ell-1)} W_V^{(\ell)}$$

dependem só dos *tokens* $1, \ldots, i$. Chegar um *token* novo não muda nenhum deles. Dá para calcular uma vez e guardar.

E a consulta? A consulta de um *token* antigo já foi usada para produzir a saída dele, lá no passo dele. Nenhuma conta futura volta a ler essa consulta. Por isso o *cache* se chama KV, e não QKV: só chaves e valores ficam guardados.

O PASSO DE GERAÇÃO COM CACHE

A geração passa a ter duas fases. No ***prefill***, o *prompt* inteiro passa pelo modelo de uma vez, em paralelo, e as chaves e os valores de todas as posições vão para o *cache*. Depois vem o ***decode***, um passo por *token*. No passo t, em cada camada e cabeça, o modelo:

1. calcula só os três vetores do *token* novo: $q_t = x_t W_Q$, $k_t = x_t W_K$, $v_t = x_t W_V$;
2. anexa k_t e v_t ao *cache*: $K \leftarrow [K;\, k_t]$ e $V \leftarrow [V;\, v_t]$;
3. faz a atenção de uma linha só, a do *token* novo:

$$o_t = \mathrm{softmax}\left(\frac{q_t\, K_{1:t}^{\top}}{\sqrt{d_k}}\right) V_{1:t}$$

Aqui nem precisa de máscara: a única consulta é a última posição, e todo o resto já é passado.

EXEMPLO RESOLVIDO: QUANTO TRABALHO O CACHE POUPA

Prompt de $P = 1.000$ *tokens* e $n = 500$ *tokens* gerados, como no código da questão. Sem *cache*, o passo t (de 0 a 499) processa $P + t$ posições:

$$\sum_{t=0}^{n-1}(P+t) = nP + \frac{n(n-1)}{2}$$

$$= 500.000 + 124.750 = 624.750$$

Com *cache*: o *prefill* processa 1.000 posições e cada um dos outros 499 passos processa 1. Total: $1.000 + 499 = 1.499$ posições. São cerca de **417 vezes menos**.

Em contas de ponto flutuante, um *forward* custa perto de $2N$ operações por posição (N é o número de parâmetros). Com um modelo de 8 bilhões de parâmetros: sem *cache*, $2 \times 8 \cdot 10^9 \times 624.750 \approx 1{,}0 \cdot 10^{16}$; com *cache*, $\approx 2{,}4 \cdot 10^{13}$.

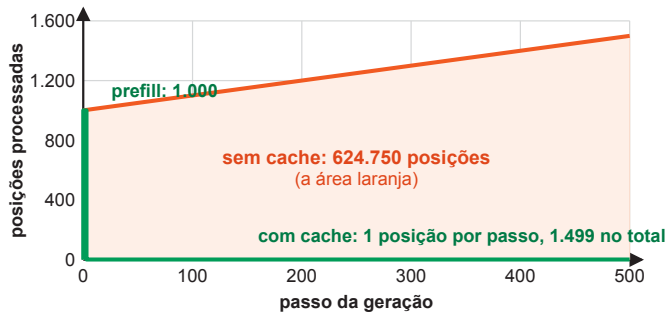


Figura 2. Sem cache, o passo t reprocessa o prompt e tudo o que já foi gerado (1.000 + t posições). Com cache, o prompt passa uma vez (prefill) e cada passo processa só o token novo. O trabalho total é a área sob cada curva.

O PREÇO: MEMÓRIA

O *cache* troca conta por memória. Para cada *token* guardado, cada camada guarda uma chave e um valor por cabeça de KV:

$$\text{bytes} = 2 \times L \times H_{kv} \times d_h \times b \times T \times B$$

onde L é o número de camadas, H_{kv} as cabeças de chave e valor, d_h a dimensão de cada cabeça, b os bytes por número, T os *tokens* e B o tamanho do lote. O 2 conta K e V.

No Llama 3 8B (32 camadas, 8 cabeças de KV, dimensão 128, 2 bytes em BF16): $2 \times 32 \times 8 \times 128 \times 2 = 131.072$ bytes, ou **128 KiB por *token***. Uma conversa de 8.192 *tokens* ocupa 1 GiB só de *cache*; cinco conversas ao mesmo tempo, 5 GiB. Se o modelo usasse as 32 cabeças de KV da atenção clássica, seriam 4 GiB por conversa. É por isso que existem a atenção de consultas agrupadas (GQA, *grouped-query attention*), que divide cada chave entre várias consultas, e o PagedAttention, que guarda o *cache* em blocos, como a memória virtual de um sistema operacional, para não desperdiçar espaço.

Há ainda um efeito na velocidade. No *decode*, cada passo faz pouca conta e precisa ler da memória os pesos e o *cache* inteiro. O gargalo vira a banda de memória, e não a capacidade de cálculo da GPU. A questão 02 explora justamente essa conta.

EM UMA FRASE

Com a máscara causal, as chaves e os valores do passado nunca mudam. O KV *cache* guarda esses vetores e cada passo calcula só o *token* novo, trocando muita conta repetida por memória.

NÃO CONFUNDA OS TRÊS CACHES

KV *cache*: dentro de uma geração, guarda K e V dos *tokens* já vistos. ***Cache* de prefixo** (ou de *prompt*): reaproveita o KV de um começo idêntico entre requisições diferentes; é o assunto da questão 03. ***Cache* semântico**: guarda a resposta pronta e a devolve para uma pergunta parecida, sem chamar o modelo.

AGORA, AS ALTERNATIVAS

A *guarda as respostas de perguntas repetidas e as devolve sem rodar o modelo de novo.*

Errada. Isso descreve um *cache* de respostas, às vezes chamado de *cache* semântico, que fica na aplicação e devolve um texto pronto sem rodar o modelo. O KV *cache* fica dentro do modelo, durante uma única geração, e guarda vetores K e V, não respostas. O modelo continua rodando a cada *token*.

B *reduz o tamanho dos pesos do modelo, mantendo na memória só as camadas mais usadas.*

Errada. O KV *cache* não toca nos pesos. Ele guarda ativações, os K e V de cada *token*, e ainda ocupa memória extra: 128 KiB por *token* no Llama 3 8B, como mostra a conta. Encolher pesos é quantização, assunto da questão 02.

C *evita recalcular as chaves (K) e os valores (V) dos* tokens *anteriores, e cada passo processa só o* token *novo.*

Certa. É o que a aula mostrou. Com a máscara causal, K e V dos *tokens* anteriores não mudam quando chega um *token* novo, em nenhuma camada. Então eles são calculados uma vez e guardados, e cada passo do *decode* calcula só q_t, k_t e v_t do *token* novo. No exemplo, o trabalho cai de 624.750 para 1.499 posições.

D *permite atenção bidirecional na geração, já que o* cache *guarda também os* tokens *futuros.*

Errada. O futuro ainda não existe durante a geração, e a atenção continua causal. É justamente a máscara causal que torna o *cache* válido: num modelo bidirecional, como o BERT, um *token* novo mudaria as representações de todos os anteriores, e o *cache* não serviria.

E *guarda as consultas (Q) dos* tokens *anteriores, que são reaproveitadas pelos* tokens *seguintes.*

Errada. É o erro mais sutil. A consulta de um *token* antigo já foi usada no passo dele e nunca mais é lida. No passo novo, só a consulta do *token* atual entra na conta, contra as chaves e os valores de todos. Por isso o *cache* guarda K e V, e não Q.

DE ONDE VEIO A PERGUNTA

"Why do you use a KV cache? (in the context of LLMs)", com a etiqueta [Google], na lista Top 24 LLM questions asked at DeepMind, OpenAI, Meta and more (BuildML, dez. 2025). É uma compilação: o autor diz que conversou com candidatos e condensou as perguntas que ouviram.

PARA ESTUDAR

- Hugging Face, Caching (como o KV cache funciona)
- Pope et al., Efficiently Scaling Transformer Inference, 2022
- Ainslie et al., GQA, 2023
- Kwon et al., PagedAttention (vLLM), 2023

QUESTÃO 02

Gabarito **E**

Um modelo de 70 bilhões numa GPU de 80 GB: cabe? e quão rápido?

ENTENDENDO O TEMA

O PROBLEMA

A empresa quer rodar um modelo de 70 bilhões de parâmetros numa única H100, com 80 GB de memória, atendendo uma conversa por vez. Antes de comprar qualquer coisa, a equipe precisa responder a duas perguntas. A primeira: o modelo cabe na GPU com folga para o resto? A segunda: quantos *tokens* por segundo dá para esperar?

Muita gente erra a segunda pergunta porque faz a conta pelo poder de cálculo da GPU, que é enorme. Nesta situação, quem manda é outra coisa: a velocidade com que a GPU consegue ler a própria memória. Vamos fazer as duas contas, uma de cada vez.

PRIMEIRA CONTA: QUANTO OCUPAM OS PESOS

Cada parâmetro é um número guardado na memória. O espaço total é o número de parâmetros N vezes os bytes de cada um, b:

$$\text{memória dos pesos} = N \times b$$

Em BF16 (*bfloat16*), cada número ocupa 2 bytes. Em INT8, 1 byte. Em INT4, meio byte. Passar de 16 para 8 ou 4 bits é a quantização (*quantization*): cada peso vira um inteiro pequeno q, e um grupo de pesos vizinhos divide uma escala s e um zero z guardados em mais bits:

$$w \approx s\,(q - z), \qquad q \in \{0, 1, \ldots, 15\} \text{ no INT4}$$

Como cada grupo de 128 pesos leva uma escala de 16 bits e um zero de 4 bits, o custo real é um pouco maior que 4 bits por peso: $4 + \frac{16+4}{128} \approx 4{,}16$. Métodos como o AWQ (*activation-aware weight quantization*) escolhem as escalas olhando as ativações, para proteger os poucos pesos que mais importam e perder pouca qualidade.

A questão exige 30 GB livres para o KV *cache* (a memória da conversa, da questão 01) e as ativações. Sobram 50 GB para os pesos.

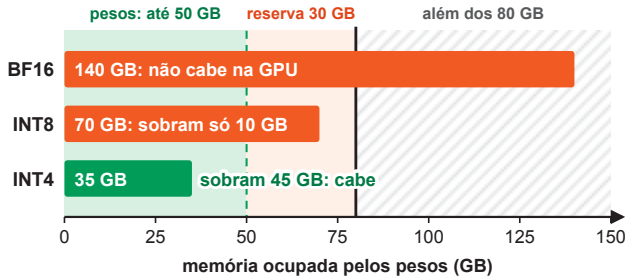


Figura 1. Os 80 GB da H100 menos os 30 GB exigidos para o KV cache e as ativações deixam 50 GB para os pesos. Só o INT4 cabe. O INT8 entra na GPU, mas come a reserva.

SEGUNDA CONTA: POR QUE A VELOCIDADE DEPENDE DA MEMÓRIA

Na geração, o modelo produz um *token* por vez. Para cada *token* novo, todos os pesos passam pelas unidades de cálculo da GPU: os 70 bilhões de números precisam sair da memória (HBM) e chegar aos núcleos. Se a GPU lê a memória a uma banda B_{mem}, há um teto que nenhum truque de software fura:

$$\text{tokens/s} \le \frac{B_{\text{mem}}}{\text{bytes lidos por token}}$$

E o cálculo? Um *forward* custa perto de $2N$ operações por *token*: 140 bilhões de operações no nosso modelo. A H100 faz cerca de 989 trilhões de operações por segundo em BF16, então o cálculo sozinho permitiria uns 7 mil *tokens* por segundo. A memória, como vamos ver, permite menos de 100. O menor dos dois limites vence.

O JEITO DE ENXERGAR: O ROOFLINE

Há uma medida que diz qual dos dois limites manda: a intensidade aritmética I, o número de operações feitas para cada byte lido da memória. O desempenho possível é

$$\text{desempenho} = \min\left(\text{pico de cálculo},\ B_{\text{mem}} \times I\right)$$

Na H100, o ponto de virada fica em $989{,}5 \div 3{,}35 \approx 295$ operações por byte. Abaixo disso, a GPU espera a memória; acima, espera o cálculo. Agora a conta do *decode*: cada peso lido (2 bytes em BF16) entra numa multiplicação e numa soma (2 operações) para cada conversa do lote. Com lote B:

$$I = \frac{2B}{2 \text{ bytes}} = B \ \text{ operações por byte}$$

Com uma conversa só, $I = 1$: quase 300 vezes abaixo do ponto de virada. O *decode* com lote 1 é limitado pela memória, e a Figura 2 mostra onde ele cai.

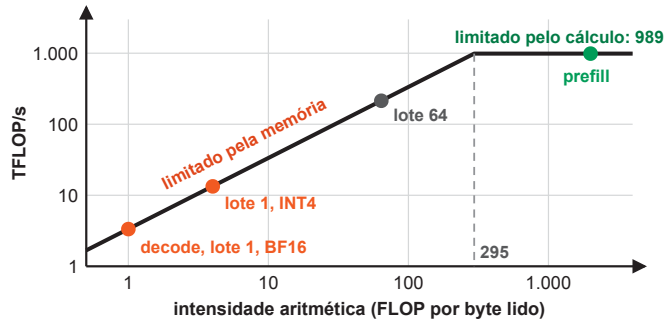


Figura 2. O desempenho possível é o menor entre o pico de cálculo e a banda vezes a intensidade aritmética. Gerando para uma conversa só, cada byte lido serve para pouquíssima conta: o ponto fica lá embaixo, na rampa da memória, quase 300 vezes abaixo do pico. Só lotes grandes ou o prefill chegam ao teto plano.

EXEMPLO RESOLVIDO: AS TRÊS OPÇÕES LADO A LADO

Banda da H100 SXM: 3,35 TB/s. Memória: 80 GB. Reserva exigida: 30 GB.

Formato	Pesos	Sobra	Cabe?	Teto
BF16	140 GB	−60 GB	não	23,9 tok/s
INT8	70 GB	10 GB	não (< 30)	47,9 tok/s
INT4	35 GB	45 GB	sim	95,7 tok/s

A coluna do teto é $3{,}35 \cdot 10^{12} \div (70 \cdot 10^{9} \times b)$. Cortar os bytes pela metade dobra o teto: a velocidade cresce porque se move menos dado, não porque a conta ficou mais rápida.

Com as escalas e as camadas de entrada e saída guardadas em 16 bits, o arquivo INT4 real do Llama 3.1 70B (AWQ) tem 39,8 GB. Ainda sobram 40 GB, e o teto fica em $3{,}35 \div 0{,}0398 \approx 84$ *tokens* por segundo.

O KV CACHE TAMBÉM É LIDO

A conta acima ignora um detalhe que a questão 01 explica: a cada passo, o modelo lê também o KV *cache* da conversa. Num modelo como o Llama 3 70B (80 camadas, 8 cabeças de KV, dimensão 128, 2 bytes), cada *token* do contexto ocupa $2 \times 80 \times 8 \times 128 \times 2 = 327.680$ bytes, ou 320 KiB. Os 30 GB de reserva guardam uns 91 mil *tokens* de contexto. E, com 8.192 *tokens* de conversa, cada passo lê $35 + 2{,}7 = 37{,}7$ GB, e o teto cai para uns 89 *tokens* por segundo. Quanto mais longa a conversa, mais lento fica cada *token*.

NA PRÁTICA

Os números são tetos teóricos. Nenhum *kernel* usa 100% da banda, e o INT4 ainda gasta um pouco para desfazer a quantização antes de cada conta. Também há um custo de qualidade, pequeno com métodos como o AWQ, que precisa ser medido na sua tarefa.

Quando o serviço tem muitas conversas, o jogo muda. Os pesos lidos uma vez servem a todas as conversas do lote no mesmo passo, e a intensidade sobe junto com o lote. É por isso que os servidores de inferência juntam pedidos (*continuous batching*): a vazão total cresce quase de graça até o lote chegar perto do ponto de virada. Para acelerar uma conversa só, as saídas são outras: mais GPUs somando banda (*tensor parallelism*) ou *speculative decoding*, que a questão 03 comenta.

EM UMA FRASE

Gerando para uma conversa por vez, cada *token* exige ler todos os pesos, então o teto de velocidade é a banda de memória dividida pelos bytes do modelo. Quantizar para INT4 faz o modelo de 70B caber com folga e quadruplica esse teto.

NÃO CONFUNDA

Quantização encolhe os pesos. **PagedAttention** organiza o KV *cache* em blocos para não desperdiçar memória; não toca nos pesos. **Teto pelo cálculo** (uns 7 mil *tokens*/s aqui) e **teto pela memória** (uns 95) são limites diferentes, e vale o menor.

AGORA, AS ALTERNATIVAS

A *pesos em BF16 (cerca de 140 GB), com teto de aproximadamente 24 tokens por segundo.*

Errada. A conta da velocidade está certa: $3{,}35\ \text{TB/s} \div 140\ \text{GB} \approx 24$ *tokens* por segundo. Mas a primeira conta falha antes: 140 GB de pesos não cabem em 80 GB de memória. A opção nem chega a rodar.

B *pesos em BF16 com PagedAttention, que reduz pela metade o espaço dos pesos, com teto de cerca de 48 tokens por segundo.*

Errada. O PagedAttention organiza o KV *cache* em blocos, como a memória virtual de um sistema operacional, para não desperdiçar espaço entre conversas. Ele não comprime pesos. Em BF16 os pesos continuam com 140 GB e continuam não cabendo.

C *pesos em INT8 (cerca de 70 GB), com teto de cerca de 48 tokens por segundo, metade do tempo de leitura em BF16.*

Errada. Os 70 GB e os 48 *tokens* por segundo estão certos, e o modelo até entra na GPU. Mas sobram só 10 GB, abaixo dos 30 GB que a questão exige para o KV *cache* e as ativações. Na Figura 1, a barra do INT8 invade a reserva.

D *pesos em INT4 (cerca de 35 GB), com teto de milhares de tokens por segundo, pois as contas em 4 bits são mais rápidas que as de 16 bits.*

Errada. O tamanho está certo, mas o motivo e o número não. Milhares de *tokens* por segundo seria o teto pelo cálculo (uns 7 mil), que só vale quando a memória não limita. Com uma conversa por vez, o *decode* fica na rampa da memória da Figura 2. O ganho do INT4 vem de mover menos bytes; a conta em si nem costuma ser feita em 4 bits.

E *pesos em INT4 (cerca de 35 GB), com espaço livre para o KV cache e teto de cerca de 95 tokens por segundo.*

Certa. É o resultado do exemplo resolvido. Em INT4, os pesos ocupam cerca de 35 GB e sobram uns 45 GB, acima dos 30 GB exigidos. Com uma conversa por vez, o teto é a banda dividida pelos bytes lidos: $3{,}35\ \text{TB/s} \div 35\ \text{GB} \approx 95$ *tokens* por segundo. É um teto teórico: com o arquivo real, as perdas dos *kernels* e o KV *cache*, sai menos.

DE ONDE VEIO A PERGUNTA

"You want to serve a 70B-parameter model on a single 80 GB GPU. Walk me through whether it fits and what single-stream tokens/sec you'd expect." Atribuída à NVIDIA numa compilação pública de relatos (Outcome School).

PARA ESTUDAR

- NVIDIA, ficha técnica da H100
- Lin et al., AWQ, 2023
- Horace He, Making Deep Learning Go Brrrr From First Principles
- Databricks, LLM Inference Performance Engineering

QUESTÃO 03

Gabarito **A**

Cache de prompt: por que o acerto ficou em 0%

ENTENDENDO O TEMA

O PROBLEMA

Um chat precisa mostrar a primeira palavra em até 3 segundos para 95% dos usuários. O prompt de sistema é grande: um manual de políticas de 8 mil *tokens* e a descrição das ferramentas, com mais 2 mil. A equipe ligou o *cache* de *prompt* do provedor, que promete pular o processamento de um começo de prompt já visto. Nada mudou: a taxa de acerto ficou em 0% e o tempo até o primeiro *token* continuou o mesmo.

Para achar o defeito, precisamos de duas peças: para onde vai o tempo antes da primeira palavra, e o que exatamente o *cache* guarda.

PARA ONDE VAI O TEMPO ATÉ O PRIMEIRO TOKEN

O tempo até o primeiro *token* (TTFT, *time to first token*) tem três partes: a espera na fila do servidor, a rede e o *prefill*, a passada em que o modelo lê o prompt inteiro de uma vez e monta o KV *cache* da conversa (questão 01). Depois disso vem o *decode*, um *token* por passo. De forma aproximada:

$$\mathrm{TTFT} \approx t_{\mathrm{fila}} + t_{\mathrm{rede}} + \frac{T_{\mathrm{prompt}} - T_{\mathrm{cache}}}{v_{\mathrm{prefill}}}$$

$$t_{\mathrm{total}} \approx \mathrm{TTFT} + (n-1)\cdot \mathrm{TPOT}$$

Aqui T_{prompt} é o número de *tokens* do prompt, T_{cache} os que vieram prontos do *cache*, v_{prefill} a velocidade do *prefill*, n os *tokens* da resposta e TPOT o tempo por *token* de saída (*time per output token*). Repare no que entra em cada conta. O tamanho da resposta só aparece no tempo total. O TTFT depende de quantos *tokens* de prompt precisam ser processados. Com 11 mil *tokens* de prompt, o *prefill* domina.

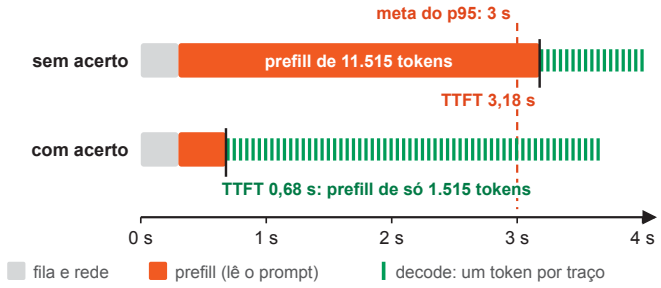


Figura 2. O primeiro token só sai depois do prefill. Com o prompt inteiro para processar, o TTFT estoura a meta; com 10 mil tokens vindos do cache, sobra folga. O decode, depois, é igual nos dois casos. Velocidade de prefill ilustrativa: 4.000 tokens por segundo.

O QUE O CACHE DE PROMPT GUARDA

O *cache* de prompt guarda o KV *cache* do começo de um prompt já processado. Se o pedido seguinte começa com os mesmos *tokens*, o servidor copia esse trecho pronto e só faz o *prefill* do resto. A saída do modelo não muda: é a mesma conta, só que reaproveitada.

A regra decisiva vem da máscara causal. A chave e o valor do *token* i dependem de todos os *tokens* de 1 até i. Se um único *token* no começo muda, todos os K e V dali para frente mudam também. Por isso o *cache* só vale para um prefixo idêntico, do primeiro *token* em diante. Servidores como o vLLM guardam o prefixo em blocos e encadeiam os blocos assim:

$$h_j = \mathrm{hash}\left(h_{j-1},\ \text{tokens do bloco } j\right)$$

O endereço de cada bloco carrega o do bloco anterior. Basta o primeiro bloco ser diferente para nenhum endereço seguinte bater. A documentação da Anthropic diz que o acerto exige trechos "100% idênticos"; a da OpenAI pede o material estável primeiro e manda pôr datas e conteúdo do usuário no fim.

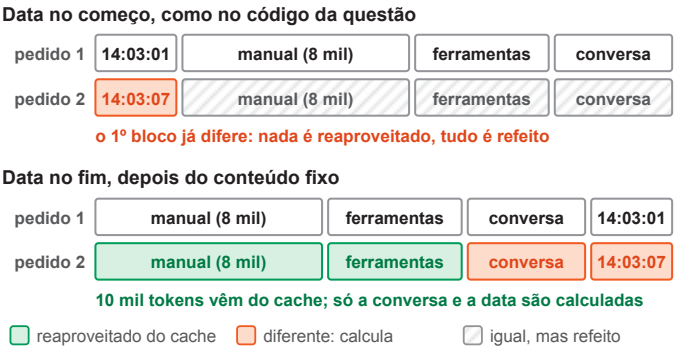


Figura 1. O cache casa o novo pedido com o antigo do começo para o fim e para no primeiro bloco diferente. Com a data na frente, o pedido muda já no primeiro bloco, e os 10 mil tokens fixos (8 mil do manual e 2 mil das ferramentas), idênticos, são refeitos mesmo assim.

O DEFEITO NO CÓDIGO DA QUESTÃO

A primeira linha do prompt é a data e a hora com segundos. Ela muda a cada pedido. Então o primeiro bloco nunca se repete, e os 10 mil *tokens* fixos que vêm depois, idênticos em todo pedido, são refeitos todas as vezes. Ligar o *cache* não muda nada, porque não existe prefixo repetido para achar. A correção é mudar a ordem: o conteúdo fixo primeiro, e o que varia, no fim.

EXEMPLO RESOLVIDO: O ANTES E O DEPOIS

Prompt de 11.515 *tokens*: 15 de data, 8.000 de manual, 2.000 de ferramentas e 1.500 de conversa. Suponha 0,3 s de fila e rede e *prefill* de 4.000 *tokens* por segundo (números ilustrativos).

Data no começo: $T_{\text{cache}} = 0$, então $\text{TTFT} \approx 0{,}3 + 11.515 \div 4.000 \approx 3{,}18$ s. Estoura a meta de 3 s.

Data no fim: $T_{\text{cache}} = 10.000$, então $\text{TTFT} \approx 0{,}3 + 1.515 \div 4.000 \approx 0{,}68$ s.

O custo cai junto. Se a leitura do *cache* custa 0,1 do preço normal de entrada, como em vários modelos atuais da Anthropic e da OpenAI, a entrada passa de 11.515 para $10.000 \times 0{,}1 + 1.515 = 2.515$ unidades: 78% mais barata. O primeiro pedido de cada janela paga a escrita, um pouco mais cara (1,25 vez o preço, na Anthropic).

NA PRÁTICA

Monte o prompt do mais estável para o mais variável: ferramentas, instruções, documentos fixos, histórico da conversa e, por último, a pergunta e tudo o que muda a cada pedido, como data, nome do usuário ou identificador da sessão. Se precisar da data, ponha no fim, ou arredonde para o dia, e mesmo assim depois do conteúdo fixo.

Três detalhes contam. Há um tamanho mínimo para entrar no *cache* (1.024 *tokens* em vários modelos). As entradas vencem: na Anthropic, o padrão é 5 minutos, renovados a cada uso. E dá para medir: as respostas das APIs informam quantos *tokens* vieram do *cache*. Se esse número fica em zero com tráfego constante, o prefixo está mudando.

EM UMA FRASE

O *cache* de prompt só reaproveita um começo idêntico, *token* por *token*. Qualquer coisa que muda no início, como a hora com segundos, apaga o acerto do resto. Ponha o fixo primeiro e o variável no fim.

NÃO CONFUNDA

Cache de prompt reaproveita o KV de um começo idêntico entre pedidos diferentes. **KV cache** (questão 01) reaproveita K e V dentro de uma mesma geração. ***Speculative decoding*** acelera os *tokens* depois do primeiro. **Cache semântico** devolve uma resposta pronta para pergunta parecida, sem rodar o modelo.

AGORA, AS ALTERNATIVAS

A *mover a data e a hora para o fim do prompt, depois do conteúdo fixo, porque o cache só reaproveita prefixos idênticos.*

Certa. É o defeito que a aula mostrou. O *cache* casa o pedido novo com o antigo do primeiro *token* em diante e para na primeira diferença, porque K e V de cada *token* dependem de tudo o que veio antes. Com a data no fim, os 10 mil *tokens* fixos formam um prefixo idêntico e saem prontos do *cache*. No exemplo, o TTFT cai de 3,18 s para 0,68 s.

B *reduzir o `max_tokens` da resposta, porque o tempo até o primeiro token cresce com o tamanho da saída.*

Errada. O `max_tokens` limita quantos *tokens* a resposta pode ter. Isso mexe no tempo total, o termo $(n-1) \cdot \text{TPOT}$, que vem depois do primeiro *token*. O TTFT acontece antes de qualquer *token* de saída e depende do *prefill* do prompt.

C *ativar o* speculative decoding*, que acelera o processamento do prompt antes do primeiro token.*

Errada. O *speculative decoding* usa um modelo pequeno para propor alguns *tokens* e o modelo grande para conferir todos numa passada só. Ele acelera o *decode*, depois do primeiro *token*, e não o processamento do prompt. O TTFT continua igual.

D *aumentar para uma hora o tempo de vida do cache, para que as entradas não expirem entre as requisições.*

Errada. As entradas não estão expirando: elas nunca casam. Como a hora muda a cada segundo, cada pedido tem um prefixo novo, e um tempo de vida maior só guardaria por mais tempo prefixos que ninguém vai repetir. Na Anthropic, a escrita com vida de uma hora ainda custa o dobro do preço normal.

E *trocar por um modelo de janela de contexto maior, para que o prompt de 10 mil tokens caiba com folga.*

Errada. O prompt de cerca de 11 mil *tokens* já cabe na janela. Uma janela maior não diminui o *prefill* nem faz o prefixo se repetir. O problema é a ordem do conteúdo, não o espaço.

DE ONDE VEIO A PERGUNTA

Atribuída à Microsoft numa compilação pública de relatos (Outcome School): "A Copilot chat feature has a p95 budget of 3 seconds to first useful content. Where does the time go, and how do you cut it?". Variante da Apple: "Time-to-first-token for your on-device feature is 1.8 s. Walk me through diagnosing and fixing it."

PARA ESTUDAR

- Anthropic, documentação de prompt caching
- OpenAI, guia de prompt caching
- vLLM, Automatic Prefix Caching (desenho)
- Leviathan et al., Speculative Decoding, 2023

QUESTÃO 04

Gabarito **B**

Temperatura 0 e não determinismo em servidores de inferência

ENTENDENDO O TEMA

O PROBLEMA

Você pede a mesma coisa mil vezes a um servidor de inferência, com temperatura 0, e espera mil respostas iguais. No experimento do texto-base, com o Qwen3-235B e o pedido "Tell me about Richard Feynman", saíram 80 respostas diferentes. A mais comum apareceu só 78 vezes. As mil respostas andam juntas por 102 *tokens*. No 103º, 992 delas escrevem "Queens, New York" e 8 escrevem "New York City". Dali em diante, cada grupo segue seu próprio caminho.

Isso incomoda quem faz avaliação, quem depura um erro e quem treina com reforço. Para entender de onde vem a variação, vamos por partes: primeiro, o que a temperatura 0 faz; depois, onde a conta pode mudar sem ninguém mexer em nada.

TEMPERATURA 0 NÃO SORTEIA

No fim de cada passo, o modelo dá uma nota, o *logit* z_i, para cada *token* possível. A temperatura T entra antes do *softmax*, que transforma as notas em probabilidades:

$$p_i(T) = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$$

Quanto menor o T, mais a probabilidade se concentra no maior *logit*. No limite, toda a massa vai para ele, e o servidor nem sorteia: escolhe direto o maior. É a decodificação gulosa (*greedy decoding*):

$$y_t = \arg\max_i z_i$$

Então, se a resposta muda, não foi sorte. Foram os próprios *logits* que mudaram de uma execução para outra. E os *logits* saem de bilhões de somas.

SOMAR EM OUTRA ORDEM DÁ OUTRO NÚMERO

O computador guarda cada número com um número fixo de dígitos binários. Depois de cada soma, o resultado é arredondado para o valor representável mais próximo:

$$\mathrm{fl}(a+b) = (a+b)(1+\delta), \qquad |\delta| \le u$$

O u é o erro máximo de um arredondamento. Ele vale cerca de $6 \cdot 10^{-8}$ em float32, $4{,}9 \cdot 10^{-4}$ em float16 e $3{,}9 \cdot 10^{-3}$ em bfloat16, o formato mais usado para servir LLMs. Como cada soma arredonda um pouco, a ordem das somas muda o resultado. A soma em ponto flutuante não é associativa:

$$\mathrm{fl}\big(\mathrm{fl}(a+b)+c\big) \neq \mathrm{fl}\big(a+\mathrm{fl}(b+c)\big)$$

Em float16, entre 1024 e 2048 só existem números inteiros. Então $(1024 + 0{,}5) + 0{,}5$ dá 1024, porque 1024,5 arredonda para 1024 duas vezes. Já $1024 + (0{,}5 + 0{,}5)$ dá 1025. Até em precisão dupla, a mais fina do Python, $(0{,}1 + 0{,}2) + 0{,}3$ dá 0,6000000000000001, e $0{,}1 + (0{,}2 + 0{,}3)$ dá 0,6.

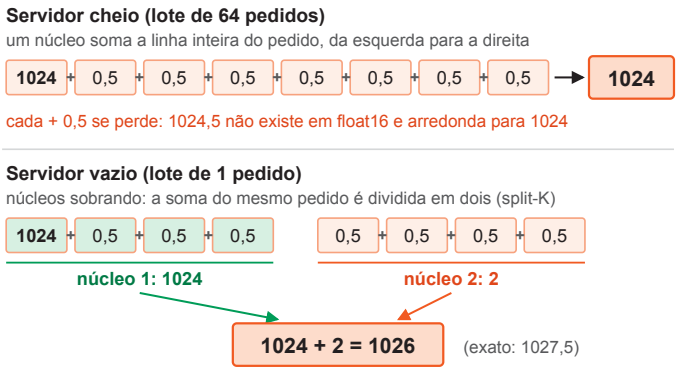


Figura 1. Os mesmos oito números, na mesma GPU, somados em float16. Com o servidor cheio, cada núcleo cuida de um pedido e soma em linha: dá 1024. Com o servidor vazio, o kernel divide a soma para não deixar núcleos parados: dá 1026. Os números foram escolhidos para a diferença aparecer; numa GPU de verdade ela costuma ficar nos últimos bits.

QUEM DECIDE A ORDEM: O TAMANHO DO LOTE

Um passo do modelo está cheio de somas longas, as reduções. Cada elemento de uma multiplicação de matrizes é um produto interno, $\sum_k x_k W_{kj}$, com milhares de termos. A normalização RMSNorm soma os quadrados do vetor inteiro. A atenção soma sobre todas as posições do KV *cache*.

O *kernel*, a rotina que faz essa conta na GPU, escolhe a estratégia pelo tamanho do lote. Com muitos pedidos juntos, cada núcleo pega um pedaço da saída e faz a soma inteira sozinho. Com poucos pedidos, sobrariam núcleos parados. Então o *kernel* divide a própria soma entre vários núcleos (*split-K*, ou *split-KV* na atenção) e junta os pedaços no fim. A mesma conta, para o mesmo pedido, passa a ser feita em outra ordem.

E o tamanho do lote depende da carga. O servidor junta no mesmo passo os pedidos de todos os usuários que chegaram naquela hora. O texto do Horace He resume assim: para um usuário, os outros usuários não são uma entrada do sistema, são uma propriedade não determinística dele. Sua resposta muda porque o lote mudou, e o lote mudou porque outras pessoas estavam usando o servidor.

Repare no que **não** é a causa. Rodar a mesma multiplicação, nos mesmos dados e com o mesmo lote, dá sempre o mesmo resultado, bit a bit. O *forward* de um LLM não usa somas atômicas, aquelas em que as *threads* chegam em ordem imprevisível. Ele é determinístico de uma execução para outra. O que falta é **invariância ao tamanho do lote** (*batch invariance*).

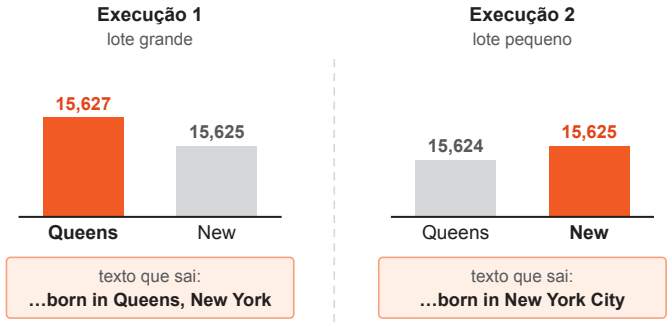


Figura 2. No token 103 do experimento, dois candidatos estão quase empatados. Com temperatura 0 ganha o maior logit. Uma diferença na terceira casa decimal, vinda da ordem das somas, troca o vencedor, e o resto do texto passa a seguir outro caminho.

EXEMPLO RESOLVIDO: UM EMPATE NA TERCEIRA CASA DECIMAL

Suponha que, no token 103, o *logit* de "Queens" seja 15,627 e o de "New" seja 15,625 (valores ilustrativos). Com $T = 1$, a chance de "Queens" seria

$$p = \frac{1}{1 + e^{-(15{,}627 - 15{,}625)}} = \frac{1}{1 + e^{-0{,}002}} \approx 0{,}5005$$

Ou seja, um empate quase perfeito. Com $T = 0$ não há sorteio: ganha "Queens", sempre que os números forem esses.

Agora o servidor está vazio, o lote muda e uma soma lá atrás acontece em outra ordem. O *logit* de "Queens" cai para 15,624. A diferença relativa é de $0{,}003/15{,}6 \approx 2 \cdot 10^{-4}$, menor que um único arredondamento em bfloat16 ($u \approx 3{,}9 \cdot 10^{-3}$). Mesmo assim, agora ganha "New". Como cada *token* seguinte é calculado a partir dos anteriores, o texto nunca mais se reencontra: um sai "Queens, New York", o outro "New York City".

NA PRÁTICA

A correção do texto-base é escrever *kernels* invariantes ao lote: cada pedido usa sempre a mesma ordem de soma, seja qual for o tamanho do lote. Na atenção, por exemplo, em vez de fixar o número de pedaços da soma, fixa-se o tamanho de cada pedaço. Com isso, as 1.000 respostas do experimento saíram idênticas. O preço apareceu no tempo. Num teste de desempenho do post, o vLLM padrão levou 26 segundos; a primeira versão determinística, 55; com um *kernel* de atenção melhorado, 42.

Vale pagar quando a reprodução exata importa: avaliações que precisam dar o mesmo número, depuração de um erro raro e aprendizado por reforço *on-policy*. Nesse último caso, com amostragem e treino batendo bit a bit, a divergência KL entre as duas políticas ficou em zero no experimento. Fora desses casos, o caminho comum é aceitar a variação e medir com várias execuções.

EM UMA FRASE

Com temperatura 0 o modelo escolhe sempre o maior *logit*, mas os *logits* dependem da ordem das somas, e essa ordem muda com o tamanho do lote, que muda com a carga do servidor.

NÃO CONFUNDA

Determinismo de execução para execução: mesma conta, mesmo lote, mesmo resultado; os *kernels* de um LLM já têm. **Invariância ao lote**: o resultado de um pedido não depende de quantos outros estão no mesmo lote; é o que falta nos servidores comuns.

AGORA, AS ALTERNATIVAS

A *amostragem aleatória residual, já que a temperatura 0 apenas reduz, mas não elimina, o sorteio entre os tokens mais prováveis.*

Errada. Com temperatura 0 não existe sorteio. Como mostra a fórmula do *softmax*, a probabilidade inteira vai para o maior *logit*, e o servidor faz a escolha gulosa, o $\arg\max$. Se a resposta mudou, foram os *logits* que mudaram, não um sorteio residual.

B *falta de invariância ao tamanho do lote nos* kernels*, pois a carga do servidor muda o lote e, com ele, a ordem das somas em ponto flutuante.*

Certa. É o mecanismo da aula. O lote depende da carga do servidor; o *kernel* escolhe a estratégia de soma pelo tamanho do lote; a ordem das somas muda o arredondamento; um *logit* quase empatado troca de lado. Com *kernels* invariantes ao lote, as 1.000 respostas saíram idênticas, o que confirma a causa.

C *concorrência entre as* threads *da GPU, que somam os valores em ordem diferente a cada execução, mesmo quando o lote é o mesmo.*

Errada. É a explicação popular, e o próprio texto-base a desmonta: na GPU, repetir a mesma multiplicação nos mesmos dados dá resultado idêntico bit a bit. O *forward* de um LLM não usa somas atômicas, então com o mesmo lote não há variação. A alternativa fala justamente do caso em que o lote é o mesmo.

D *reutilização de respostas guardadas no cache de prompt, que devolve respostas antigas sempre que o prefixo coincide.*

Errada. O *cache* de *prompt* reaproveita o processamento de um prefixo, isto é, as chaves e os valores já calculados (questões 01 e 03). Ele não guarda nem devolve respostas prontas, e o modelo continua gerando cada *token*.

E *atualização silenciosa dos pesos do modelo, feita pelo provedor ao longo do dia, entre uma execução e outra.*

Errada. O experimento usa o mesmo servidor e o mesmo modelo, em execuções seguidas. E a prova está no próprio texto: trocando só os *kernels* por versões invariantes ao lote, as mil respostas ficaram iguais. Se os pesos mudassem no meio, isso não teria acontecido.

DE ONDE VEIO A PERGUNTA

"Compare greedy, beam search, top-k, top-p and temperature sampling. When does each fail?" Atribuída a Google DeepMind e Apple numa compilação pública de relatos (Outcome School). Variante na Aced, atribuída a Anthropic e OpenAI: "How do you adjust the creativity of generative AI models?".

PARA ESTUDAR

- Horace He, Defeating nondeterminism in LLM inference (2025)
- Thinking Machines, batch-invariant-ops (código)
- Goldberg, What every computer scientist should know about floating-point arithmetic

QUESTÃO 05

Gabarito **C**

LSH: hashes que fazem o parecido colidir

ENTENDENDO O TEMA

O PROBLEMA

São 50 milhões de vetores de 256 dimensões. Comparar uma consulta com todos custa $5 \cdot 10^7 \times 256 \approx 1{,}3 \cdot 10^{10}$ multiplicações, e é por isso que a busca exata leva uns 2 segundos. O produto pede menos de 50 milissegundos, cerca de 40 vezes menos. Não dá para comparar com todos. A saída é olhar só uma parte da base e aceitar, de vez em quando, perder um vizinho. É o que fazem os índices de vizinhos aproximados (ANN, *approximate nearest neighbors*).

Cada família escolhe "a parte" de um jeito. O HNSW anda por um grafo, a PQ compara versões comprimidas dos vetores e o LSH (*Locality-Sensitive Hashing*) separa a base em baldes com funções de *hash*. Para entender o LSH, comece pelo *hash* que você já conhece.

UM HASH AO CONTRÁRIO

Uma tabela de *hash* comum acha **o igual** em um passo: a chave vira um endereço. As funções de *hash* criptográficas, como o SHA-256, vão além e fazem questão de espalhar: mudar um único caractere da entrada muda metade dos bits da saída, o efeito avalanche. Ótimo para senhas e para dividir carga entre servidores. Péssimo para achar **o parecido**, porque dois vetores quase iguais vão parar em baldes sem relação nenhuma.

O LSH inverte o objetivo: ele quer colisões, desde que sejam entre itens parecidos. Indyk e Motwani definiram isso em 1998. Uma família de funções h é sensível à localidade quando, para uma distância r e um fator $c > 1$,

$$d(q,x) \le r \;\Rightarrow\; \Pr[h(q) = h(x)] \ge p_1$$
$$d(q,x) \ge c\,r \;\Rightarrow\; \Pr[h(q) = h(x)] \le p_2$$

com $p_1 > p_2$. Em português: quem está perto colide com probabilidade alta; quem está longe, com probabilidade baixa. O sorteio está na escolha da função h, não nos dados.

HIPERPLANOS ALEATÓRIOS: UM BIT POR CORTE

Para a similaridade de cosseno, a família mais usada é a de Charikar (2002), o SimHash. Sorteie um vetor $\mathbf{r}$ com coordenadas gaussianas. Ele define um hiperplano que passa pela origem, e o *hash* diz de que lado o vetor está:

$$h_{\mathbf{r}}(\mathbf{x}) = \begin{cases} 1, & \mathbf{r}\cdot\mathbf{x} \ge 0 \\ 0, & \mathbf{r}\cdot\mathbf{x} < 0 \end{cases}$$

Dois vetores $\mathbf{x}$ e $\mathbf{y}$, separados por um ângulo θ, só recebem bits diferentes se o hiperplano passar entre eles. Olhe o plano que contém os dois: o hiperplano corta esse plano numa reta, e como a gaussiana não prefere direção nenhuma, essa reta pode ter qualquer inclinação com a mesma chance. Ela separa os dois só quando cai dentro do ângulo θ, o que acontece com probabilidade θ/π. Então:

$$\Pr\left[h_{\mathbf{r}}(\mathbf{x}) = h_{\mathbf{r}}(\mathbf{y})\right] = 1 - \frac{\theta(\mathbf{x},\mathbf{y})}{\pi}$$

Vetores a 30° concordam num bit com chance $1 - 30/180 \approx 0{,}83$; vetores perpendiculares, com chance 0,5, a de uma moeda.

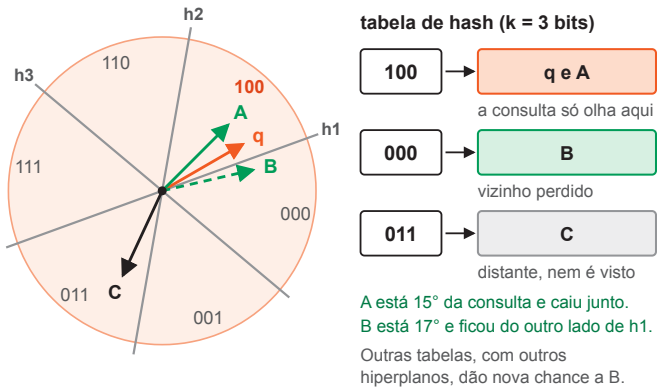


Figura 1. Cada hiperplano aleatório h_i dá um bit: 1 de um lado, 0 do outro. Três bits cortam o plano em seis baldes. A consulta q é comparada só com quem está no balde dela. O vizinho A é achado; o B, quase tão perto, caiu do outro lado de h_1 e fica de fora desta tabela.

O MECANISMO: K BITS POR TABELA, L TABELAS

Um bit só separa pouco: metade da base cai em cada lado. Por isso o LSH combina hashes de dois jeitos.

E (*AND*). Junte k bits num código, que é o endereço do balde. São 2^k baldes. Dois vetores só caem juntos se concordarem nos k bits, o que acontece com probabilidade p^k, com $p = 1 - \theta/\pi$. Isso derruba os falsos candidatos, mas também afasta vizinhos de verdade, como o B da Figura 1.

OU (*OR*). Monte L tabelas, cada uma com hiperplanos sorteados de novo. Um vetor vira candidato se colidir com a consulta em pelo menos uma delas:

$$P(\text{candidato}) = 1 - \left(1 - p^k\right)^L$$

A consulta calcula os $k \cdot L$ bits, junta os vetores dos L baldes dela, mede a distância exata só desses candidatos e devolve os melhores. A fórmula dá a curva em S da Figura 2. O degrau fica perto da similaridade em que $p \approx (1/L)^{1/k}$. Com $k = 10$ e $L = 20$, isso dá $p \approx 0{,}74$, um ângulo perto de 47°.

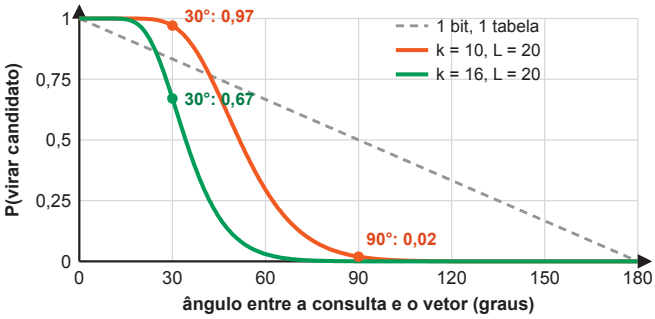


Figura 2. A curva em S. Com um bit só, a chance de colidir cai em linha reta com o ângulo. Juntando k bits (E) e L tabelas (OU), a curva vira um degrau: quem está perto quase sempre vira candidato, quem está longe quase nunca. Mais bits empurram o degrau para ângulos menores.

EXEMPLO RESOLVIDO: O VIZINHO A 30° E O ESTRANHO A 90°

Com $k = 10$ bits e $L = 20$ tabelas. Um vizinho a 30° da consulta:

$$p = 1 - \tfrac{30}{180} \approx 0{,}833, \qquad p^{10} \approx 0{,}162$$
$$P = 1 - (1 - 0{,}162)^{20} \approx \mathbf{0{,}97}$$

Um vetor sem relação, a 90°:

$$p = 0{,}5, \qquad p^{10} \approx 0{,}00098$$
$$P = 1 - (1 - 0{,}00098)^{20} \approx \mathbf{0{,}019}$$

Se quase toda a base estiver longe, uns 90° da consulta, sobram cerca de $5 \cdot 10^7 \times 0{,}019 \approx 970$ mil candidatos. É 2% da base, uns 50 vezes menos trabalho que a busca exata. Quer menos candidatos? Suba para $k = 16$: a 90° a chance cai para 0,0003, e sobram uns 15 mil. Mas o vizinho a 30° passa a ser achado só 67% das vezes. Para recuperar, aumente L, e aí entra a memória: cada tabela guarda um id para cada vetor. Com 20 tabelas, são $5 \cdot 10^7 \times 20 \times 4$ bytes, 4 GB só de ids.

NA PRÁTICA E OS LIMITES

O LSH dá uma garantia de probabilidade, não de resultado. Um vizinho de verdade pode cair do lado errado em todas as tabelas, como o B da figura. Dá para deixar esse risco tão pequeno quanto se queira, mas pagando com mais tabelas, isto é, mais memória.

Há outro limite: os hiperplanos são sorteados sem olhar os dados. Em *embeddings* densos, que se concentram em regiões do espaço, índices que aprendem com os dados costumam render mais recall pelo mesmo custo. No ANN-Benchmarks, os métodos de grafo, como o HNSW, ficam na frente na maioria das bases, e o IVF do Faiss vai bem. A própria documentação do Faiss diz que o LSH perde para a PQ em memória e velocidade para a mesma precisão. Cuidado com o nome: o `IndexLSH` do Faiss guarda códigos binários e compara todos pela distância de Hamming, sem baldes nem tabelas.

Onde o LSH brilha é na deduplicação de texto, com o MinHash, a versão para conjuntos: a chance de dois documentos terem o mesmo MinHash é a similaridade de Jaccard entre seus pedaços de texto. O GPT-3 usou o MinHashLSH do Spark, com 10 hashes, para tirar documentos quase repetidos do treino. O FineWeb usou 112 hashes em 14 grupos de 8, mirando pares com cerca de 75% de semelhança.

EM UMA FRASE

O LSH sorteia funções de *hash* que fazem vetores parecidos colidirem com alta probabilidade; a consulta só compara com quem caiu nos mesmos baldes, e k e L regulam a troca entre velocidade, recall e memória.

NÃO CONFUNDA OS QUATRO ÍNDICES

LSH: baldes por *hash* sorteado. **HNSW**: grafo em camadas, busca gulosa de cima para baixo. **IVF**: *k-means* divide a base em listas e a consulta abre só as mais próximas. **PQ**: comprime cada vetor trocando pedaços dele pelo id do centróide mais próximo.

AGORA, AS ALTERNATIVAS

A *Um grafo em camadas em que a busca desce do nível mais alto, com poucos nós, até o mais baixo, com todos.*

Errada. Esse é o HNSW (*Hierarchical Navigable Small World*): um grafo em camadas, com poucos nós no topo, em que a busca desce gulosa até a camada de baixo, que tem todos. O LSH não usa grafo nenhum; usa funções de *hash* e baldes.

B *Uma compressão que divide cada vetor em subvetores e guarda só o identificador do centróide mais próximo de cada um.*

Errada. Essa é a quantização por produto (PQ, *product quantization*): cada vetor é cortado em subvetores, e cada pedaço vira o id do centróide mais próximo num dicionário aprendido. É compressão, não *hash*. O LSH não aprende centróides; sorteia hiperplanos.

C *Funções de* hash *feitas para que vetores próximos caiam no mesmo balde com alta probabilidade; a consulta só é comparada com os vetores dos seus baldes.*

Certa. É a definição da aula. A família de *hash* é escolhida para que $\Pr[h(q) = h(x)]$ seja alta quando x está perto da consulta e baixa quando está longe, como no SimHash, em que a chance é $1 - \theta/\pi$. A consulta só mede a distância exata dos vetores que caíram nos mesmos baldes dela, em uma das L tabelas.

D *Uma busca exata que usa* hashes *para garantir 100% de* recall *com menos comparações.*

Errada. O LSH é aproximado, e a garantia é de probabilidade. No exemplo, com $k = 10$ e $L = 20$, um vizinho a 30° fica de fora 3% das vezes; com $k = 16$, 33%. Dá para reduzir o risco com mais tabelas, nunca zerar de graça. Busca exata com 100% de *recall* é comparar com todos.

E *Um* hash *criptográfico, como o SHA-256, que espalha vetores parecidos por baldes distantes para equilibrar a carga.*

Errada. É o contrário do LSH. Um *hash* criptográfico tem efeito avalanche: vetores quase iguais ganham *hashes* sem relação. Isso serve para espalhar carga ou conferir integridade, mas destrói exatamente a vizinhança que a busca precisa preservar.

DE ONDE VEIO A PERGUNTA

"Explain Locality-sensitive hashing (LHS) indexing method?", assim mesmo, com a sigla trocada, na seção "Internal Working of Vector Databases" do repositório LLMInterviewQuestions (GitHub). É uma compilação de perguntas de entrevista sobre LLMs, sem atribuição a uma empresa.

PARA ESTUDAR

- Andoni e Indyk, Near-optimal hashing algorithms for approximate nearest neighbor (CACM, 2008)
- Charikar, Similarity estimation techniques from rounding algorithms (2002)
- Leskovec, Rajaraman e Ullman, Mining of Massive Datasets, cap. 3
- Faiss, guia dos índices

QUESTÃO 06

Gabarito **D**

Busca híbrida: o embedding entende o sentido, o BM25 lê a palavra exata

ENTENDENDO O TEMA

O PROBLEMA

A loja tem uma busca semântica que funciona bem quando o cliente descreve o que quer, como "tênis leve para corrida". Mas falha justo no caso mais fácil para um humano: o cliente digita o código do fabricante, "XJ-4471", e o produto certo aparece só em 38º. Na frente dele vêm produtos parecidos.

O texto dá uma pista de ouro: trocar o índice aproximado (HNSW) pela busca exata não mudou nada. Então o índice não está perdendo o item, como pode acontecer com os índices da questão 05. A busca exata compara a consulta com os 2 milhões de vetores, e mesmo assim o item certo fica em 38º. O defeito está na **representação**: para o modelo de *embedding*, o produto certo não é o vizinho mais próximo de "XJ-4471".

DOIS JEITOS DE MEDIR "PARECIDO"

Um **modelo de *embedding*** transforma cada texto num vetor denso, com centenas de números, treinado para que textos de mesmo sentido fiquem perto. A nota é o cosseno entre o vetor da consulta $\mathbf{q}$ e o do produto $\mathbf{d}$:

$$\cos(\mathbf{q}, \mathbf{d}) = \frac{\mathbf{q} \cdot \mathbf{d}}{\|\mathbf{q}\| \, \|\mathbf{d}\|} \in [-1, 1]$$

Isso é ótimo para sinônimo e paráfrase. Mas um código como "XJ-4471" não tem sentido. O modelo o quebra em pedaços ("XJ", "44", "71") e o coloca perto de outros códigos parecidos e de outros fones. Trocar um dígito quase não move o vetor, e para o cliente esse dígito é tudo.

A **busca lexical** (*lexical search*) faz o contrário: conta palavras. O padrão da área é o BM25. Para uma consulta com termos t e um documento D:

$$\mathrm{BM25}(D,Q) = \sum_{t \in Q} \mathrm{IDF}(t) \cdot \frac{f(t,D)\,(k_1+1)}{f(t,D) + k_1\,\beta_D}$$

$$\beta_D = 1 - b + b\,\frac{|D|}{\mathrm{avgdl}}$$

Aqui $f(t,D)$ é quantas vezes o termo aparece no documento, $|D|$ é o tamanho do documento e avgdl, o tamanho médio; β_D só junta essa correção de tamanho. O k_1 (em geral entre 1,2 e 2,0) faz a contagem saturar: repetir uma palavra ajuda cada vez menos. O b (em geral 0,75) desconta documentos longos. E o IDF (*inverse document frequency*) dá peso à raridade:

$$\mathrm{IDF}(t) = \ln\left(\frac{N - n(t) + 0{,}5}{n(t) + 0{,}5} + 1\right)$$

com N documentos no total e $n(t)$ documentos que contêm o termo. Um código que só existe num produto é o termo mais raro possível, e por isso pesa muito.

EXEMPLO: O PESO DA RARIDADE

Com $N = 2.000.000$ produtos, um código que aparece em um só produto tem $\mathrm{IDF} = \ln(1.333.334) \approx 14{,}1$. A palavra "tênis", presente em, digamos, 40 mil produtos, tem $\mathrm{IDF} = \ln(50) \approx 3{,}9$. O código vale **3,6 vezes mais** que a palavra comum. Já a saturação: com $k_1 = 1{,}2$ e tamanho médio, o fator de contagem vale 1,0 para uma ocorrência, 1,37 para duas e 1,96 para dez. Nunca passa de $k_1 + 1 = 2{,}2$.

JUNTAR AS DUAS LISTAS SEM MISTURAR AS RÉGUAS

A ideia da **busca híbrida** (*hybrid search*) é rodar as duas buscas e juntar os resultados. O cuidado está no "juntar". As duas notas vivem em escalas diferentes: o cosseno fica entre −1 e 1, e o BM25 não tem teto fixo, pois cresce com o número de termos da consulta e com a raridade deles. Somar as duas é somar metros com quilômetros (Figura 2).

A saída mais usada é a ***Reciprocal Rank Fusion*** (RRF). Ela ignora as notas e usa só a posição $r(d)$ de cada documento em cada lista:

$$\mathrm{RRF}(d) = \sum_{\text{listas}} \frac{1}{k + r(d)}, \qquad k = 60$$

Quem não aparece numa lista ganha zero dela. Como só posições entram na conta, a escala das notas deixa de importar. O $k = 60$ amacia o topo: o 1º lugar vale $1/61$ e o 2º, $1/62$, quase o mesmo. Assim, uma lista sozinha não decide tudo, e quem vai bem nas duas sobe. Os autores fixaram $k = 60$ num estudo piloto e notaram que o valor exato pouco importa.

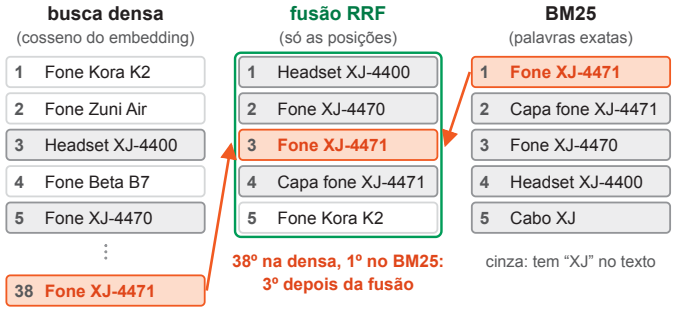


Figura 1. A consulta é "XJ-4471". A busca densa acha produtos parecidos e deixa o item exato em 38º. O BM25 acha o código, que só aparece no texto dele, e o põe em 1º. A fusão soma 1/(60 + posição) de cada lista: quem aparece bem nas duas sobe, e o item exato pula para 3º.

EXEMPLO RESOLVIDO: A FUSÃO PASSO A PASSO

As posições são as da Figura 1. Para o item exato:

$$\mathrm{RRF}(\text{XJ-4471}) = \frac{1}{60+38} + \frac{1}{60+1}$$
$$= 0{,}0102 + 0{,}0164 = 0{,}0266$$

Para o *headset* XJ-4400, 3º na densa e 4º no BM25: $1/63 + 1/64 = 0{,}0159 + 0{,}0156 = 0{,}0315$.

Produto	Densa	BM25	RRF
Headset XJ-4400	3º	4º	0,0315
Fone XJ-4470	5º	3º	0,0313
Fone XJ-4471	38º	1º	**0,0266**
Capa fone XJ-4471	120º	2º	0,0217
Fone Kora K2	1º	fora	0,0164

O item exato sai do 38º para o 3º. Repare que a fusão premia o consenso: quem está bem nas duas listas passa na frente de quem está em 1º numa só. Para códigos, é comum somar um atalho simples: se a consulta tem cara de código, procurar primeiro a correspondência exata no campo do código. Com ele, o XJ-4471 vai para o 1º lugar.

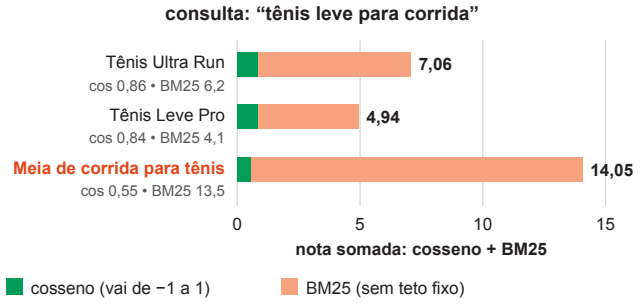


Figura 2. Somar as notas cruas mistura réguas diferentes. O cosseno mal passa de 1, e o BM25 chega a dezenas. A soma vira quase só BM25, e a meia que repete "corrida" e "tênis" passa na frente dos tênis de corrida. Valores ilustrativos.

NA PRÁTICA

Busca híbrida virou o padrão em RAG e em busca de produtos. No estudo de *Contextual Retrieval* da Anthropic, juntar *embeddings* e BM25 (os dois com contexto) baixou a taxa de falha da busca no top 20 de 5,7% para 2,9%, uma queda de 49%. Com um *reranker* depois, caiu para 1,9%. O *reranker* é o assunto da questão 07.

A RRF também tem limites. Ela joga fora a força da nota: um 1º lugar folgado vale o mesmo que um 1º lugar apertado. Por isso alguns sistemas normalizam as notas de cada lista e fazem uma média com pesos, mas aí é preciso escolher a normalização e calibrar o peso com dados rotulados. A RRF é o ponto de partida que funciona sem calibrar nada.

EM UMA FRASE

O *embedding* acha o sentido e erra o termo exato; o BM25 acha o termo exato e não entende sentido. A busca híbrida roda as duas e junta as listas pelas posições, com a RRF, sem misturar as réguas das notas.

NÃO CONFUNDA

***Recall* do índice**: quanto um índice aproximado acha dos vizinhos que a busca exata acharia. **Relevância**: se os vizinhos mais próximos são os itens certos. Trocar HNSW por busca exata só mexe no primeiro. Aqui o problema era o segundo.

AGORA, AS ALTERNATIVAS

A *somar a nota do BM25 diretamente ao cosseno do* embedding *e ordenar os produtos pela soma das duas notas.*

Errada. É a armadilha das réguas. O cosseno vai de −1 a 1 e o BM25 chega a dezenas, então a soma vira quase só BM25. O código até subiria, mas as buscas descritivas, que hoje são boas, quebrariam: na Figura 2, a meia que repete "corrida" passa na frente dos tênis. Somar só funciona depois de normalizar e calibrar pesos, e a RRF evita esse trabalho.

B *trocar o modelo de* embedding *por outro com mais dimensões, capaz de representar melhor sequências de caracteres.*

Errada. Mais dimensões dão mais espaço para representar sentido, e o problema não é falta de sentido. Um código não tem sentido para ser representado. O modelo continuaria aproximando "XJ-4471" de códigos e produtos parecidos. Correspondência exata é trabalho da busca lexical.

C *fazer* fine-tuning *do modelo de linguagem que redige a resposta, usando o catálogo completo.*

Errada. O erro está antes da geração. O modelo que redige a resposta só lê o que a busca entrega, e o item certo está em 38º, fora do que chega a ele. Ajustar quem escreve não muda o que foi recuperado.

D *adicionar uma busca lexical (BM25) e fundir as duas listas pelas posições, com* Reciprocal Rank Fusion.

Certa. É a busca híbrida da aula. O BM25 pega o código, que é raro e por isso tem IDF alto, e a RRF junta as duas listas pelas posições, sem misturar as escalas das notas. No exemplo, o item exato sai do 38º para o 3º, e um atalho de correspondência exata no campo do código o leva ao 1º.

E *aumentar de 5 para 50 o número de vizinhos devolvidos pela busca vetorial e exibir todos ao usuário.*

Errada. Com 50 resultados, o item em 38º passaria a aparecer, mas enterrado. Isso empurra o trabalho para o cliente e não conserta a ordem. Quem digita um código espera o produto em primeiro.

DE ONDE VEIO A PERGUNTA

Relato direto de candidato a Forward Deployed Engineer na OpenAI (por volta de mai/2026): o *take-home* foi "Build a semantic search setup over Amazon products for ChatGPT" (Aced). Atribuída à Microsoft em compilação: "How do you choose between a sparse retriever (BM25) and a dense retriever? When do you need both?"

PARA ESTUDAR

- Cormack, Clarke e Büttcher, Reciprocal Rank Fusion, SIGIR 2009
- Robertson e Zaragoza, The Probabilistic Relevance Framework: BM25 and Beyond, 2009
- Anthropic, Contextual Retrieval

QUESTÃO 07

Gabarito **B**

Reranker: ele reordena os candidatos, não acha os que faltaram

ENTENDENDO O TEMA

O PROBLEMA

O sistema de RAG da questão busca em duas etapas. Primeiro um *bi-encoder* traz 50 trechos. Depois os 8 primeiros vão para o modelo de linguagem, que escreve a resposta. A equipe mediu a busca com 400 perguntas rotuladas, cada uma com um único trecho que contém a evidência, e os números não animaram: a evidência está entre os 8 que o modelo lê em só 48% das perguntas.

A proposta é pôr um *reranker* do tipo *cross-encoder* sobre os 50. A pergunta de entrevista por trás é: o que um *reranker* consegue consertar, e o que ele não alcança? Para responder, precisamos ler as métricas com cuidado e entender por que existem dois tipos de modelo.

O QUE AS MÉTRICAS DIZEM

Chame de r_q a posição do trecho com a evidência na lista da pergunta q. O **Recall@k** conta em quantas perguntas a evidência apareceu entre os k primeiros:

$$\text{Recall@}k = \frac{1}{|Q|}\sum_{q\in Q}\mathbf{1}\left[r_q \le k\right]$$

O **MRR@k** (*mean reciprocal rank*) olha a posição: vale $1/r_q$ se a evidência está no top k e zero se não está. Em 1º lugar, vale 1; em 4º, 0,25.

$$\text{MRR@}k = \frac{1}{|Q|}\sum_{q\in Q}\frac{\mathbf{1}\left[r_q \le k\right]}{r_q}$$

Agora leia o quadro da questão com essas definições. Recall@50 = 0,62 quer dizer que, em 38% das perguntas, a evidência nem está entre os 50 candidatos. Recall@8 = 0,48 quer dizer que em 14% ela está nos 50, mas abaixo do 8º lugar, e o modelo não a lê. E o MRR@8 de 0,31 mostra que, mesmo quando chega, ela costuma chegar longe do topo.

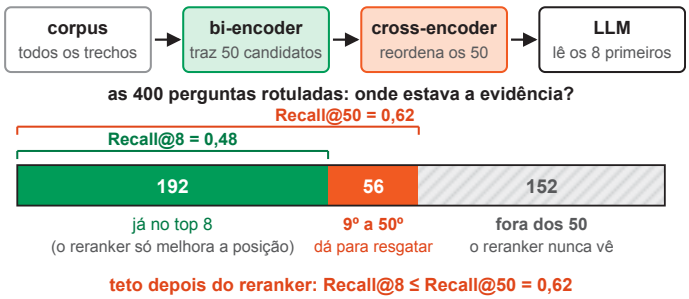


Figura 1. O reranker trabalha só com os 50 candidatos que a primeira etapa entregou. Das 400 perguntas, 192 já tinham a evidência no top 8 e 56 a tinham entre o 9º e o 50º lugar: são essas que ele pode resgatar. As 152 cuja evidência ficou fora dos 50 continuam perdidas.

A DESIGUALDADE QUE DECIDE A QUESTÃO

Um *reranker* recebe os 50 candidatos e devolve os mesmos 50 em outra ordem. Ele não busca nada no resto da base. Então o conjunto dos 50 não muda, e o Recall@50 também não. E os 8 primeiros depois da reordenação são sempre 8 dos 50. Daí sai uma regra que vale para qualquer *reranker*, do mais simples ao melhor do mundo:

$$\text{Recall@8}_{\text{depois}} \;\leq\; \text{Recall@50}_{\text{antes}} = 0{,}62$$

No melhor caso, o *reranker* põe a evidência em 1º lugar em todas as 248 perguntas em que ela está entre os 50. Aí o Recall@8 vai a 0,62, e o MRR@8 também, porque cada uma dessas perguntas passa a valer 1. Passar disso é impossível sem mexer na primeira etapa.

POR QUE EXISTEM DOIS MODELOS

Se o *cross-encoder* é mais preciso, por que não usá-lo desde o começo? Por causa do custo. No ***bi-encoder***, consulta e trecho passam pelo modelo separados, e cada um vira um vetor. A nota é o produto interno $\mathbf{q} \cdot \mathbf{d}$. Como o vetor do trecho não depende da pergunta, todos os trechos são processados uma vez, na indexação. Na hora da pergunta, só ela passa pelo modelo, e um índice vetorial acha os vizinhos.

No ***cross-encoder***, consulta e trecho entram juntos no mesmo modelo, e a atenção cruza as palavras dos dois. Ele percebe coisas que um vetor resumido perde, como uma negação ou qual entidade faz o quê. Mas a nota depende do par, então nada pode ser preparado antes. O custo por pergunta cresce com o número de candidatos N:

$$t_{\text{consulta}} \approx t_{\text{busca}} + N \cdot t_{\text{par}}$$

O artigo do Sentence-BERT dá a escala: achar o par mais parecido entre 10.000 frases exige cerca de 50 milhões de passadas com o BERT lendo os pares juntos, umas 65 horas. Com vetores separados, leva cerca de 5 segundos. Por isso a receita é em duas etapas: o *bi-encoder* filtra a base inteira barato, e o *cross-encoder* capricha na ordem de poucos candidatos.

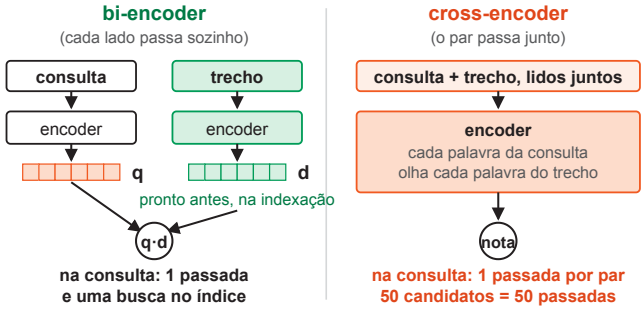


Figura 2. No bi-encoder, o vetor de cada trecho é calculado uma vez, antes de qualquer pergunta; na consulta, só a pergunta passa pelo modelo. No cross-encoder, a nota depende do par, então nada pode ser calculado antes: é uma passada do modelo para cada candidato.

EXEMPLO RESOLVIDO: O QUE MUDA E O QUE NÃO MUDA

Cinco perguntas, com a posição da evidência antes e depois do *reranker*:

Pergunta	Antes	Depois
P1	1º	1º
P2	3º	1º
P3	12º	2º
P4	fora dos 50	fora dos 50
P5	6º	1º

$$\text{MRR@8}_{\text{antes}} = \tfrac{1}{5}\left(1 + \tfrac{1}{3} + 0 + 0 + \tfrac{1}{6}\right) = 0{,}30$$
$$\text{MRR@8}_{\text{depois}} = \tfrac{1}{5}\left(1 + 1 + \tfrac{1}{2} + 0 + 1\right) = 0{,}70$$

O Recall@8 sobe de 3/5 = 0,6 para 4/5 = 0,8, porque a P3 foi resgatada do 12º lugar. O Recall@50 fica em 0,8 antes e depois: a P4 continua perdida, e o teto do Recall@8 é justamente esse 0,8.

No caso da questão, o ganho máximo é de 56 perguntas (14% de 400): o Recall@8 pode ir de 0,48 até 0,62, nunca além.

NA PRÁTICA

O *reranker* vale a pena quando a evidência está nos candidatos, mas em posição ruim. No estudo de *Contextual Retrieval* da Anthropic, a busca trazia 150 candidatos e o *reranker* escolhia os 20 melhores. A taxa de falha no top 20 caiu de 2,9% para 1,9%. Repare que o recall do topo subiu, como na alternativa certa, porque havia evidências entre o 21º e o 150º lugar para resgatar. O próprio texto avisa da troca: reordenar mais candidatos melhora a qualidade, mas aumenta a latência e o custo.

Para passar do teto de 0,62, o conserto é na primeira etapa: busca híbrida com BM25 (questão 06), *chunking* melhor, reescrita da consulta ou mais candidatos, pagando o custo linear do *reranker*. E vale medir as duas etapas separadas: Recall@50 para a busca e MRR ou nDCG para a ordem final.

EM UMA FRASE

O *reranker* só reordena o que a busca trouxe: melhora a posição e o Recall do topo até o teto do Recall dos candidatos, nunca acima dele, e custa uma passada do modelo por candidato.

NÃO CONFUNDA

Recall@50 mede a primeira etapa: a evidência chegou aos candidatos? **Recall@8 e MRR@8** medem o que o modelo de linguagem lê. O *reranker* mexe nos dois últimos e deixa o primeiro igual.

AGORA, AS ALTERNATIVAS

A *elevará o Recall@50, porque o* cross-encoder *lê a consulta e o trecho juntos e encontra documentos que a busca deixou passar.*

Errada. O Recall@50 depende só de quais 50 trechos a busca trouxe, e o *reranker* devolve os mesmos 50 em outra ordem. As 152 perguntas cuja evidência ficou de fora nunca chegam até ele. É verdade que o *cross-encoder* lê consulta e trecho juntos, mas só dos pares que recebe; ele não procura nada na base.

B *pode subir o Recall@8 até, no máximo, 0,62, já que só reordena os candidatos e não alcança as consultas cuja evidência ficou fora dos 50.*

Certa. É a desigualdade da aula: depois de reordenar, os 8 primeiros são 8 dos mesmos 50, então o Recall@8 só pode subir até o Recall@50 de 0,62. O ganho possível são as 56 perguntas com a evidência entre o 9º e o 50º lugar.

C *deve substituir o* bi-encoder *na indexação de todo o corpus, pois é mais preciso e pode ser calculado com antecedência.*

Errada. É o contrário. A nota do *cross-encoder* depende da consulta, então não há nada para calcular com antecedência. Usá-lo no corpus inteiro seria uma passada do modelo para cada trecho da base, a cada pergunta. É por isso que o *bi-encoder*, que deixa os vetores prontos, faz a primeira etapa.

D *reduzirá a latência da busca, já que o* cross-encoder *processa cada trecho mais rápido que o* bi-encoder.

Errada. Ele é mais lento: faz uma passada completa do modelo para cada par de consulta e trecho, e essa conta vem somada à busca. A latência total sobe.

E *terá custo fixo por consulta, independentemente da quantidade de candidatos que precisar reordenar.*

Errada. O custo cresce com o número de candidatos: $N \cdot t_{\text{par}}$. Reordenar 100 em vez de 50 custa o dobro. É a troca que a Anthropic descreve entre reordenar mais candidatos e manter a latência baixa.

DE ONDE VEIO A PERGUNTA

"What is a reranker, when should you use one, and what does a cross-encoder cost you?" Atribuída à Microsoft (e a Cohere e Perplexity) em compilação (Outcome School).

PARA ESTUDAR

- Nogueira e Cho, Passage Re-ranking with BERT, 2019
- Reimers e Gurevych, Sentence-BERT, 2019
- Anthropic, Contextual Retrieval

QUESTÃO 08

Gabarito **E**

Por que o modelo chuta com confiança, e como mudar isso

ENTENDENDO O TEMA

O PROBLEMA

O Texto I descreve um modelo que, em temas sensíveis, responde com segurança coisas erradas. Ele não diz "não tenho certeza". Ele inventa uma data, uma dose, um artigo de lei, e inventa com o mesmo tom de quem sabe. A pergunta da entrevista é como investigar e corrigir isso.

O Texto II dá a pista com uma comparação que todo mundo viveu: a prova de múltipla escolha que não desconta o erro. Nela, deixar em branco vale zero e chutar pode valer um ponto. Quem quer nota chuta. O artigo de Kalai e colegas defende que os modelos de linguagem estão o tempo todo nesse "modo prova", e que é a régua usada para treiná-los e compará-los que os empurra para o chute.

DE ONDE VEM O ERRO, E DE ONDE VEM O CHUTE

São duas perguntas diferentes. A primeira é por que o modelo erra. No pré-treino, ele aprende a estatística do texto, e alguns fatos aparecem uma vez só nos dados. Para esses, não há padrão para aprender. O artigo mostra que a taxa de alucinação do modelo base deve ser pelo menos a fração de fatos que aparecem uma única vez nos dados: se 20% dos aniversários de pessoas aparecem só uma vez nos dados, espere errar pelo menos uns 20% dessas perguntas. Algum erro é inevitável.

A segunda pergunta é o que o modelo faz quando não sabe. Aqui entra a avaliação. Na maioria dos testes, a nota é binária: 1 se acerta, 0 se erra e 0 se responde "não sei". Se o modelo acha que a resposta certa tem probabilidade p, a nota esperada de cada atitude é:

$$\mathbb{E}[\text{responder}] = p \cdot 1 + (1-p) \cdot 0 = p \;\geq\; 0 = \mathbb{E}[\text{"não sei"}]$$

Responder nunca perde. Com qualquer p acima de zero, ganha. O artigo chama isso de Observação 1: com nota binária, se abster (*abstain*) nunca é a melhor escolha. Treine e compare modelos com essa régua e o vencedor será o que chuta.

A RÉGUA QUE MUDA O COMPORTAMENTO

A proposta do artigo é dizer na própria instrução qual é o alvo de confiança t: "responda só se tiver mais de t de confiança; o erro custa $t/(1-t)$ pontos, o acerto vale 1 e o 'não sei' vale 0". Agora a conta muda:

$$\mathbb{E}[\text{responder}] = p - (1-p)\,\frac{t}{1-t}$$
$$> 0 \iff p(1-t) > (1-p)\,t \iff p > t$$

Responder só compensa quando a confiança passa do limiar. Com $t = 0{,}5$, o erro custa 1 ponto; com $t = 0{,}75$, custa 3; com $t = 0{,}9$, custa 9. O limiar $t = 0$ é a nota binária de sempre. O modelo que só responde acima do limiar e diz "não sei" abaixo dele é o melhor para qualquer t, desde que o limiar esteja escrito. É o que os autores chamam de calibração comportamental (*behavioral calibration*).

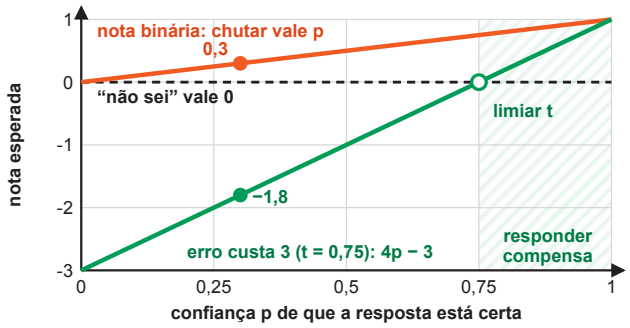


Figura 1. Nota esperada de responder, contra a confiança p. Na nota binária (laranja), responder vale p e nunca perde para o "não sei" (zero). Com penalidade 3 por erro (verde), responder só compensa acima de p = 0,75, a faixa hachurada. Com p = 0,3, o chute vale 0,3 numa régua e −1,8 na outra.

POR QUE ISSO VIRA HÁBITO DO MODELO

O artigo olhou dez avaliações muito usadas para comparar modelos, como GPQA, MMLU-Pro, SWE-bench e HLE. Nove dão nota binária e nenhum crédito ao "não sei"; só uma (WildBench) dá crédito parcial. Os autores então comparam dois modelos. O A é honesto: sinaliza a incerteza e nunca alucina. O B é igual, mas nunca admite dúvida e chuta sempre. Nessas avaliações, o B ganha. Como os placares orientam o pós-treino e as escolhas das equipes, o chute confiante é recompensado em cada etapa. Por isso a conclusão é sociotécnica: não basta criar mais uma avaliação de alucinação; é preciso mudar a nota das avaliações principais, para que parem de punir a incerteza.

EXEMPLO RESOLVIDO: O MESMO MODELO, DUAS RÉGUAS

Uma pergunta. O modelo acha que acerta com $p = 0{,}3$. Na nota binária, responder vale $0{,}3$ e o "não sei" vale 0: ele responde. Com $t = 0{,}75$, o erro custa $0{,}75/0{,}25 = 3$ pontos:

$$0{,}3 \times 1 - 0{,}7 \times 3 = 0{,}3 - 2{,}1 = -1{,}8 < 0$$

Agora o "não sei" é a melhor resposta. Com $p = 0{,}9$, responder vale $0{,}9 - 0{,}1 \times 3 = 0{,}6$, e ele responde. No limiar, $p = 0{,}75$, dá exatamente zero.

Cem perguntas. Os dois modelos sabem 80 respostas. Nas outras 20, o A diz "não sei" e o B chuta, com 25% de sorte: acerta 5 e erra 15.

Régua	A (admite)	B (chuta)
nota binária	80	**85**
erro custa 1 ($t = 0{,}5$)	**80**	85 − 15 = 70
erro custa 3 ($t = 0{,}75$)	**80**	85 − 45 = 40

Na régua binária, o modelo que alucina 15 vezes fica em primeiro. Basta descontar o erro para a ordem virar.

NA PRÁTICA: A RESPOSTA DA ENTREVISTA

A pergunta pede investigar e mitigar, então a resposta completa tem duas metades (Figura 2). Primeiro, investigar: juntar os casos de erro confiante, dos testes internos e dos especialistas, e separar os tipos. O modelo não sabia o fato? Recebeu a fonte certa e leu errado? A fonte estava errada? Cada tipo pede um remédio diferente; o segundo e o terceiro, por exemplo, são problema de recuperação e de documento, não de incentivo.

Depois, mitigar a causa que o Texto II aponta. Escrever na avaliação um limiar alto para temas de risco ($t = 0{,}9$ faz o erro custar 9), aceitar "não sei" e "não encontrei isso na fonte" como respostas válidas, treinar o modelo contra essa régua e medir três números separados: acerto, erro e abstenção. Um modelo que nunca erra porque nunca responde também não serve; o placar precisa mostrar as duas coisas. Buscar documentos (RAG) ajuda o modelo a saber mais, mas o artigo lembra que, com nota binária, ele continua premiado por chutar quando a busca não acha nada.

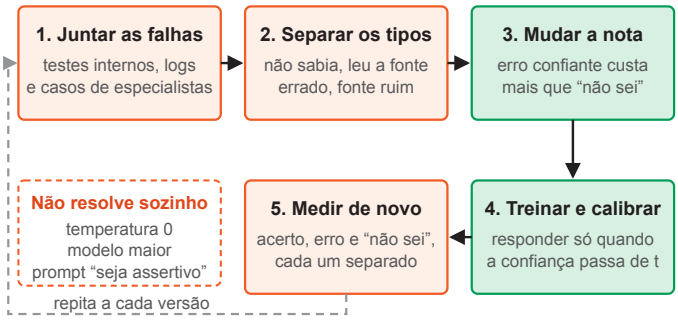


Figura 2. A resposta de entrevista em cinco passos. Antes de corrigir, investigue: junte as falhas e separe os tipos de erro. A mudança que ataca a causa é a da régua (passo 3), e o modelo é treinado e medido contra ela.

EM UMA FRASE

O modelo chuta com confiança porque a régua binária faz o chute valer mais que o "não sei"; a correção é mudar a régua, descontando o erro confiante e aceitando a abstenção, e treinar e medir o modelo contra ela.

NÃO CONFUNDA

Determinístico não é honesto. Temperatura 0 faz o modelo dar sempre a mesma resposta (questão 04), inclusive a mesma resposta errada. **Acurácia não é calibração.** Acurácia mede quanto ele acerta; calibração mede se a confiança dele bate com a chance real de acertar.

AGORA, AS ALTERNATIVAS

A *zerar a temperatura de amostragem, eliminando a aleatoriedade que produz os erros confiantes.*

Errada. Temperatura 0 escolhe sempre o *token* mais provável. Se o modelo não sabe a data e a mais provável tem 30% de chance, ele vai dizer essa data, sempre, no mesmo tom. Tira a variação entre execuções (questão 04), mas não muda o incentivo de responder em vez de admitir a dúvida. O erro só fica mais repetível.

B *ampliar o* fine-tuning *com mais exemplos do domínio, avaliados apenas pela taxa de acerto.*

Errada. Taxa de acerto é a nota binária da aula: o "não sei" vale zero e o chute vale p. Treinar mais com essa régua reforça exatamente o chute. Mais exemplos do domínio podem fazer o modelo saber mais coisas, mas não o ensinam a admitir o que não sabe.

C *substituir o modelo por um maior, cujo conhecimento cubra mais temas sensíveis.*

Errada. Um modelo maior sabe mais fatos, mas sempre sobram os raros, que aparecem uma vez só nos dados, e diante deles a mesma régua continua premiando o chute. É o caso do modelo B do artigo: chutar sempre o leva ao topo do placar binário, seja qual for o tamanho.

D *acrescentar ao prompt uma instrução para que o modelo responda sempre de forma assertiva.*

Errada. Piora o problema. Pedir assertividade é pedir para esconder a incerteza, o oposto da calibração. O modelo vai soar ainda mais seguro justamente quando não sabe.

E *penalizar o erro confiante mais do que a abstenção na avaliação e aceitar o "não sei" como resposta válida.*

Certa. Ataca a causa que o Texto II aponta. Com o erro custando $t/(1-t)$ e o "não sei" valendo zero, responder só compensa quando a confiança passa de t. No exemplo, o chute com $p = 0{,}3$ passa de 0,3 para −1,8, e o modelo honesto volta ao topo do placar. Na resposta completa, isso vem depois de juntar as falhas e separar os tipos de erro.

DE ONDE VEIO A PERGUNTA

Relato direto de candidato a Machine Learning Engineer na Anthropic, rodada de painel (por volta de out/2025). A pergunta do Texto I é a do relato, traduzida (Aced).

PARA ESTUDAR

- Kalai et al., Why Language Models Hallucinate, 2025

QUESTÃO 09

Gabarito **A**

O juiz que concorda 96% e não serve

ENTENDENDO O TEMA

O PROBLEMA

Antes de lançar um assistente, a equipe precisa saber se as respostas estão boas. Ler mil respostas à mão a cada versão não escala, então ela usa um LLM como juiz (*LLM-as-a-judge*): outro modelo lê cada resposta e diz "boa" ou "ruim". Antes de confiar nele, a equipe fez o certo: pediu a um especialista que rotulasse as mesmas mil respostas e comparou. O juiz concordou com o especialista em 96% dos casos.

Parece ótimo. Mas a pergunta que decide o lançamento não é "quantas vezes os dois concordam". É "o juiz encontra as respostas ruins?". São as ruins que podem barrar o lançamento. Para ver isso, o quadro precisa ser lido célula por célula.

A MATRIZ DE CONFUSÃO

Chame de positivo o que se quer encontrar: a resposta ruim. Cada resposta cai numa de quatro células. Verdadeiro positivo (VP): ruim, e o juiz disse ruim. Falso negativo (FN): ruim, e o juiz deixou passar. Falso positivo (FP): boa, e o juiz reprovou. Verdadeiro negativo (VN): boa, e o juiz aprovou. Com elas saem três medidas diferentes. A concordância é o acerto geral. O acerto nas ruins tem vários nomes: sensibilidade, *recall* ou taxa de verdadeiros positivos (TPR, *true positive rate*). O acerto nas boas é a especificidade, ou taxa de verdadeiros negativos (TNR).

$$\text{concordância} = \frac{VP+VN}{N} = \frac{960}{1.000} = 0{,}96$$
$$\text{TPR} = \frac{VP}{VP+FN} = \frac{6}{30} = 0{,}20$$
$$\text{TNR} = \frac{VN}{VN+FP} = \frac{954}{970} \approx 0{,}984$$

O juiz acerta quase todas as boas e só 1 em cada 5 ruins.

	o que o juiz (LLM) disse: "boa"	o que o juiz (LLM) disse: "ruim"
especialista: boa (970 respostas)	954 concordam	16 alarme falso
especialista: ruim (30 respostas)	24 ruins que passaram	6 ruins achadas

acerto nas ruins = 6 / 30 = 20%
acerto nas boas = 954 / 970 = 98,4%
concordância = (954 + 6) / 1.000 = 96%

Figura 1. O quadro da questão como matriz de confusão. Cada linha é o que o especialista disse; cada coluna, o que o juiz disse. A célula laranja forte é o problema: das 30 respostas ruins, 24 passaram como boas.

POR QUE 96% ENGANA

A concordância é uma média das duas taxas, pesada pela proporção de cada classe. Com π igual à fração de respostas ruins:

$$\begin{aligned}\text{concordância} &= \pi\,\text{TPR} + (1-\pi)\,\text{TNR} \\ &= 0{,}03 \times 0{,}20 + 0{,}97 \times 0{,}984 \approx 0{,}96\end{aligned}$$

Como só 3% das respostas são ruins, o acerto nelas pesa 3% na conta. O juiz poderia achar todas as ruins ou nenhuma, e a concordância mal se mexeria. Leve ao extremo: um "juiz" que diz "boa" para tudo, sem ler nada, concorda em 970 de 1.000, ou 97%, mais do que o juiz da questão. Quando a falha é rara, a concordância bruta mede quase só a classe comum.

DESCONTANDO O ACASO: O KAPPA DE COHEN

O kappa de Cohen (κ) corrige a concordância pelo que se esperaria só por acaso, dadas as proporções de cada lado. p_o é a concordância observada e p_e a esperada por acaso:

$$\kappa = \frac{p_o - p_e}{1 - p_e}$$

Vale 1 quando a concordância é perfeita e 0 quando não passa do acaso. É uma boa medida de resumo, mas a decisão continua dependendo das duas taxas, vistas em separado.

EXEMPLO RESOLVIDO: O KAPPA DESTE JUIZ

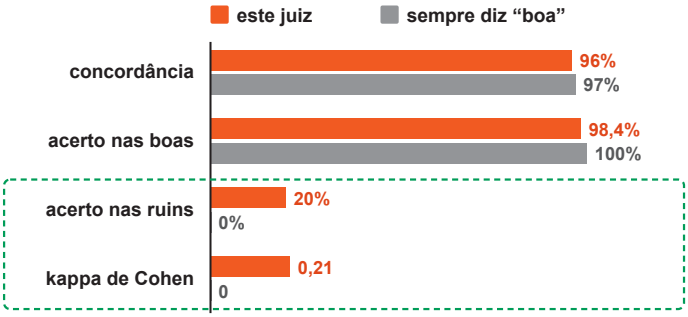
O especialista diz "boa" em 970 e "ruim" em 30. O juiz diz "boa" em $954 + 24 = 978$ e "ruim" em $16 + 6 = 22$. Se os dois respondessem ao acaso com essas proporções, concordariam em:

$$p_e = 0{,}970 \times 0{,}978 + 0{,}030 \times 0{,}022$$
$$= 0{,}94866 + 0{,}00066 = 0{,}94932$$

A concordância observada é $p_o = 0{,}96$. Então:

$$\kappa = \frac{0{,}96 - 0{,}94932}{1 - 0{,}94932} = \frac{0{,}01068}{0{,}05068} \approx 0{,}21$$

Quase toda a concordância de 96% já viria do acaso. Para o juiz que sempre diz "boa": $p_o = 0{,}97$ e $p_e = 0{,}97 \times 1 + 0{,}03 \times 0 = 0{,}97$, logo $\kappa = 0$. O kappa coloca os dois no lugar certo: o juiz da questão é um pouco melhor que nada, e o constante é nada.



Nas duas primeiras, o juiz inútil ganha ou empata.
Nas duas de baixo, no tracejado, aparece o que importa.

Figura 2. Concordância e acerto nas boas são dominados pelas 970 respostas boas, e um juiz que nunca reprova nada vence. O acerto nas ruins e o kappa, que desconta a concordância por acaso, mostram que este juiz quase não enxerga as falhas.

NA PRÁTICA: COMO VALIDAR UM JUIZ

Um juiz é um classificador e se valida como um. Separe um conjunto rotulado por especialistas e garanta falhas suficientes nele, nem que seja reforçando de propósito os casos ruins na amostra. Com só 30 ruins, o 20% medido é impreciso: o intervalo de confiança de 95% vai de cerca de 10% a 37%. Meça o acerto nas ruins e nas boas em separado, ajuste o critério do juiz (decisão binária, "passa" ou "não passa", com exemplos de falhas reais) e meça de novo num conjunto que não foi usado para ajustar. Teste também os vieses conhecidos de juízes LLM: preferir a resposta que aparece primeiro (posição), a mais longa (verbosidade) e a do próprio modelo (autopreferência, *self-enhancement*).

Com as duas taxas medidas, dá até para corrigir a taxa de falhas que o juiz relata. O juiz marca como ruins $m = 22/1.000 = 2{,}2\%$ das respostas, mas a taxa real é 3%. A correção de Rogan e Gladen recupera o valor:

$$\hat{\pi} = \frac{m + \mathrm{TNR} - 1}{\mathrm{TPR} + \mathrm{TNR} - 1}$$
$$= \frac{0{,}022 + 0{,}984 - 1}{0{,}20 + 0{,}984 - 1} \approx 0{,}030$$

Repare no denominador: com acerto nas ruins de 20%, ele fica perto de zero, e qualquer erro de medida explode na estimativa. Mais um motivo para não lançar com esse juiz.

EM UMA FRASE

Quando a falha é rara, a concordância bruta mede quase só a classe comum; um juiz se valida pelo acerto nas ruins e nas boas, em separado, contra rótulos de especialista, e este acha só 20% das ruins.

NÃO CONFUNDA

Concordância é o acerto geral, puxado pela classe maior. **Acerto nas ruins** (sensibilidade, *recall*) diz quantas falhas o juiz encontra. **Precisão** diz, entre as que ele reprovou, quantas eram mesmo ruins: aqui, 6 de 22, ou 27%.

AGORA, AS ALTERNATIVAS

A *é inadequado para a decisão, pois identifica só 20% das respostas ruins, e um juiz que sempre dissesse "boa" concordaria mais com o especialista.*

Certa. É o que a aula mostrou. O juiz marca como ruins só 6 das 30 respostas ruins, um acerto de 20%. Um juiz que dissesse "boa" para tudo concordaria em 97%, mais que os 96% deste. O kappa confirma: 0,21 contra 0 do juiz constante. Para decidir um lançamento, o que importa é achar as falhas, e esse juiz deixa passar 4 em cada 5.

B *é adequado para a decisão, porque 96% de concordância supera com folga o limiar usual de 90% exigido de juízes automáticos.*

Errada. Não existe um limiar universal de 90% que valide um juiz. A concordância depende da proporção de respostas ruins: com 97% de boas, até o juiz que não lê nada passa dos 90%. Aqui, os 96% perdem para os 97% desse chute constante.

C *ficaria adequado se passasse a dar notas de 1 a 5, adotando a média das notas como a métrica principal.*

Errada. Notas de 1 a 5 são menos estáveis entre execuções e entre avaliadores do que uma decisão binária, e a média esconde justamente as falhas raras. Mesmo que as 30 ruins recebessem nota 1 e as outras 970 nota 5, a média cairia só de 5,00 para 4,88. Trocar a escala não ensina o juiz a enxergar as ruins.

D *teria o problema resolvido com temperatura 0, que remove o viés do julgamento.*

Errada. Temperatura 0 tira a variação entre execuções (questão 04), não o viés. O juiz continuaria deixando passar as mesmas 24 respostas ruins, só que sempre as mesmas.

E *é adequado, desde que seja o mesmo modelo que gera as respostas, pois assim conhece o estilo avaliado.*

Errada. Juiz do mesmo modelo que gera as respostas tende a favorecer as próprias respostas, a autopreferência (*self-enhancement bias*) descrita por Zheng e colegas, e compartilha os mesmos pontos cegos. Conhecer o estilo é exatamente o problema, não a solução.

DE ONDE VEIO A PERGUNTA

"What methods or metrics do you use to evaluate how well your LLM performs?" Atribuída à Microsoft na Aced (por volta de 2025, lista de ML Engineer). Atribuída à Amazon em compilação: "How would you decide an LLM-powered assistant is ready to launch to millions of customers?"

PARA ESTUDAR

- Hamel Husain, Using LLM-as-a-Judge
- Zheng et al., Judging LLM-as-a-Judge, 2023

QUESTÃO 10

Gabarito **D**

Muitas ferramentas: catálogo, busca sob demanda e governança

ENTENDENDO O TEMA

O PROBLEMA

Um agente usa ferramentas assim: o modelo lê a definição de cada uma (nome, descrição e o esquema dos parâmetros) e decide qual chamar. O MCP (*Model Context Protocol*) padroniza esse encaixe: cada sistema interno vira um servidor que anuncia as suas ferramentas, e qualquer agente compatível pode usá-las. Ficou fácil ligar 200 ferramentas. O difícil começa depois.

Com 950 *tokens* por definição, as 200 ocupam 190 mil *tokens* em toda requisição. Pior que o custo é a escolha: o modelo precisa achar a ferramenta certa no meio de 200 parecidas, como `consultar_pedido` e `consultar_pedido_legado`. E algumas ações não têm volta, como estornar um pagamento. A pergunta da entrevista pede as três respostas: como achar a ferramenta (descoberta), como escolher a certa (seleção) e quem pode usar o quê (governança).

A IDEIA: FERRAMENTA TAMBÉM SE BUSCA

A saída é tratar as ferramentas como documentos de uma base, do mesmo jeito que o RAG trata textos. Todas ficam num **catálogo** indexado. O agente começa cada conversa com poucas ferramentas fixas (as três a cinco mais usadas) e uma ferramenta especial, `buscar_ferramentas`. Quando precisa de algo, descreve o que quer fazer, a busca devolve as definições mais parecidas e só elas entram no contexto. É o que a Anthropic chama de *tool search*, com as ferramentas marcadas para carregamento adiado (*deferred loading*). A OpenAI dá a mesma orientação: começar cada turno com menos de 20 funções e buscar o resto.

Repare que a seleção vira um funil de duas etapas. A busca precisa trazer a ferramenta certa f^* entre as k primeiras, e o modelo precisa escolhê-la entre essas k:

$$P(\text{acerto}) = \underbrace{P\big(f^* \in \text{top-}k\big)}_{\text{recall@}k\text{ da busca}} \times \underbrace{P\big(\text{escolhe } f^* \mid f^* \in \text{top-}k\big)}_{\text{escolha do modelo}}$$

Colocar as 200 no contexto deixa o primeiro fator em 1, mas derruba o segundo: são 199 concorrentes. A busca faz o contrário, aceita um pequeno risco no primeiro fator para deixar o segundo alto. O k regula a troca: pequeno demais, a certa fica de fora; grande demais, a confusão volta.

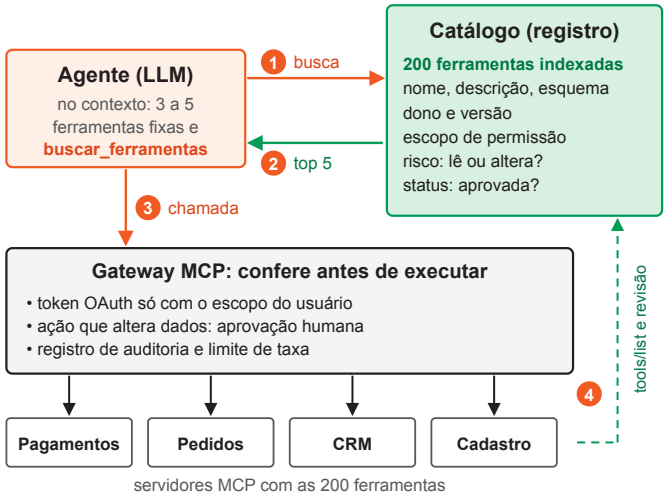


Figura 1. O agente começa com poucas ferramentas e uma de busca (1). O catálogo devolve as definições mais relevantes (2). A chamada passa por um gateway que confere escopo, pede aprovação para o que altera dados e registra tudo (3) antes de chegar ao servidor (4). O catálogo é alimentado pela lista de cada servidor, depois de revisada.

GOVERNANÇA: ACHAR NÃO É PODER USAR

Ser encontrada na busca não dá permissão a ninguém. Por isso cada ficha do catálogo guarda mais que a descrição: tem um **dono** (a equipe que responde por ela), uma **versão**, um **escopo** de permissão, um **nível de risco** (só lê ou altera dados) e o status de **aprovação** para entrar no catálogo. A especificação do MCP ajuda nas três frentes:

1. **Permissão.** A autorização segue o OAuth 2.1, com escopos pelo princípio do menor privilégio, e a lista de ferramentas pode variar conforme o escopo de quem pede. Quem não pode estornar nem vê a ferramenta de estorno.
2. **Risco.** As anotações `readOnlyHint` e `destructiveHint` dizem se a ferramenta só lê ou se apaga e altera. A especificação avisa que elas não são confiáveis se o servidor não for confiável; quem garante o risco é a revisão do catálogo.
3. **Controle humano.** A especificação recomenda que haja sempre uma pessoa capaz de negar uma chamada sensível, que os argumentos sejam mostrados antes e que todo uso fique registrado para auditoria.

A revisão também protege contra o envenenamento de ferramentas (*tool poisoning*), descrito pela Invariant Labs em 2025: instruções escondidas na descrição de uma ferramenta, que o modelo lê e obedece. Como a descrição entra no contexto, ela precisa passar por revisão e ter a versão fixada, como qualquer código.

EXEMPLO RESOLVIDO: A CONTA DOS TOKENS

Com todas as definições no contexto:

$$200 \times 950 = 190.000 \text{ tokens por requisição}$$

Com a busca, entram a ferramenta de busca (cerca de 500 *tokens*) e as cinco definições escolhidas:

$$500 + 5 \times 950 = 5.250 \text{ tokens}$$

$$1 - \frac{5.250}{190.000} \approx 97\% \text{ a menos}$$

Em 10 mil requisições por dia, são 1,9 bilhão de *tokens* contra 52,5 milhões. E o *cache* de *prompt*? A leitura de um prefixo guardado custa 0,1 do preço normal na API da Anthropic. Os 190 mil viram o custo de 19 mil, ainda 3,6 vezes os 5.250. E a janela continua 95% ocupada.

Agora o funil, com números de exemplo: se a busca acerta o top 5 em 96% das tarefas e o modelo escolhe a certa entre cinco em 92% das vezes, o acerto total é $0{,}96 \times 0{,}92 \approx 0{,}88$. Os dois fatores se medem separados, num conjunto de tarefas rotuladas com a ferramenta certa.

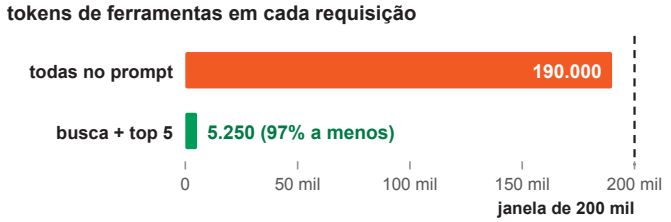


Figura 2. Com 950 tokens por definição, as 200 ferramentas ocupam 95% de uma janela de 200 mil tokens antes de o usuário dizer qualquer coisa. Com a busca, entram a ferramenta de busca (cerca de 500 tokens) e as cinco definições escolhidas.

O QUE JÁ FOI MEDIDO

A Anthropic mostrou que cinco servidores MCP comuns, com 58 ferramentas, somam cerca de 55 mil *tokens* de definições, perto de 950 por ferramenta, o número da questão. Com a busca, o exemplo do artigo caiu de cerca de 77 mil para cerca de 8,7 mil *tokens*, e o acerto na escolha subiu (Figura 3). O estudo RAG-MCP, que recupera as descrições por similaridade, mais que triplicou o acerto e cortou quase pela metade os *tokens* do *prompt*. No teste de estresse do mesmo estudo, com até 11.100 ferramentas, o acerto passava de 90% abaixo de 30 ferramentas e a falha dominava acima de cerca de 100.

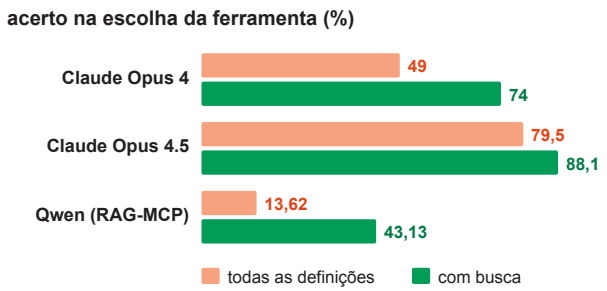


Figura 3. Medidas publicadas. Nos dois primeiros pares, a Anthropic testou o Claude com e sem a busca de ferramentas (2025). No terceiro, o estudo RAG-MCP (Gan e Sun, 2025) comparou dar todas as descrições ao modelo com recuperar só as relevantes.

NA PRÁTICA

Escreva as descrições pensando na busca: o que a ferramenta faz, quando usar e quando **não** usar. "Só pedidos anteriores a 2022" resolve boa parte da confusão entre `consultar_pedido` e `consultar_pedido_legado`, e ferramentas duplicadas devem ser aposentadas. Dê nomes únicos, com o prefixo do servidor, para evitar colisão. Meça o funil em separado: o *recall*@k da busca e o acerto da escolha. Reavalie toda vez que uma ferramenta entra ou muda. O registro oficial de servidores MCP (*MCP Registry*) só guarda servidores públicos; para ferramentas internas, a empresa mantém o próprio registro.

EM UMA FRASE

Com centenas de ferramentas, o agente busca num catálogo e carrega só as poucas relevantes, e cada ferramenta tem dono, versão, escopo e nível de risco, com aprovação humana para o que altera dados.

NÃO CONFUNDA

Descoberta é achar a ferramenta; **autorização** é poder usá-la. Uma busca boa sem escopo deixa qualquer agente estornar pagamentos. E ***cache* de *prompt*** barateia um contexto repetido, mas não o deixa menor nem mais fácil de ler.

AGORA, AS ALTERNATIVAS

A *mandar as 200 definições em todo* prompt*, contando com o* cache *de* prompt *para reduzir o custo do contexto grande.*

Errada. O *cache* de *prompt* barateia e acelera a leitura de um prefixo repetido, mas os 190 mil *tokens* continuam ocupando 95% da janela e o modelo continua escolhendo entre 200 ferramentas, que é onde ele erra. Na conta da aula, mesmo com o *cache* o custo fica 3,6 vezes o da busca. Pior: na API da Anthropic, mudar qualquer definição de ferramenta invalida o *cache* inteiro, então cada ferramenta nova ou alterada zera a economia. E a alternativa não diz nada sobre governança.

B *escolher as 10 ferramentas mais usadas da semana e mandar só essas em todo* prompt*, revisando a lista toda segunda-feira.*

Errada. Resolve o tamanho do contexto, mas quebra o primeiro fator do funil: a ferramenta rara que uma tarefa precisa simplesmente não está lá, e o agente não tem como achá-la. A lista manual da segunda-feira envelhece durante a semana e ninguém responde por ela. E também não há escopo nem dono.

C *fundir as 200 ferramentas numa só ferramenta genérica, com um parâmetro que diz qual sistema interno chamar.*

Errada. A ferramenta genérica esconde 200 comportamentos atrás de um parâmetro de texto. O modelo perde as descrições e os esquemas de cada uma, então o problema de escolher continua, só que agora dentro do parâmetro e sem validação. E a governança fica impossível: uma permissão só cobre tudo, e um estorno parece igual a uma consulta.

D *manter um catálogo indexado, buscar e carregar sob demanda só as ferramentas relevantes e dar a cada uma dono e escopo de permissão.*

Certa. É o desenho da aula. O catálogo indexado com busca resolve a descoberta, o carregamento sob demanda mantém o contexto pequeno e a escolha fácil (de 190 mil para cerca de 5 mil *tokens*), e dono e escopo de permissão por ferramenta dão a governança. Na prática, ainda entram versão, nível de risco, aprovação humana para ações que alteram dados e registro de uso.

E *fazer* fine-tuning *do modelo com as 200 ferramentas, para que ele as chame sem precisar das definições no contexto.*

Errada. As ferramentas mudam: entram novas, versões trocam de parâmetros. Cada mudança exigiria treinar de novo, e o modelo chamaria a versão que decorou. O MCP existe justamente para o agente descobrir as ferramentas na hora, pela lista do servidor. E *fine-tuning* não dá governança: permissão é conferida no servidor, não na memória do modelo.

DE ONDE VEIO A PERGUNTA

Questão 63 do post The 100 Questions That Haunt AI Engineering Interviews 2026 (AI Engineering Insider, out. 2026), na parte sobre agentes e uso de ferramentas, marcada como avançada: "Your company wants to expose 200 internal tools to agents through a standard protocol such as MCP. Including all tool definitions in every prompt is impossible. How do you design tool discovery, selection, and governance?". É uma compilação, sem empresa citada; o nível Staff é classificação nossa.

PARA ESTUDAR

- Anthropic, Introducing advanced tool use (tool search), 2025
- Anthropic, documentação da tool search tool
- Gan e Sun, RAG-MCP, 2025
- Especificação do MCP: ferramentas (2026-07-28)
- Invariant Labs, tool poisoning no MCP, 2025

QUESTÃO 11 • BÔNUS

Gabarito **B**

Prompt, RAG ou fine-tuning: fato vai no contexto, comportamento vai nos pesos

ENTENDENDO O TEMA

O PROBLEMA

A seguradora fez quatro pedidos de uma vez. Os preços mudam toda semana. Cada resposta precisa citar a tabela e a data de vigência. Cada vendedor só pode ver as condições da sua região. E as respostas seguem um formato fixo de três blocos. Parece um pedido só, mas são dois tipos de necessidade misturados. Os três primeiros falam de **informação**: o que é verdade hoje, de onde veio e quem pode ver. O último fala de **comportamento**: como responder. Separar os dois resolve a questão.

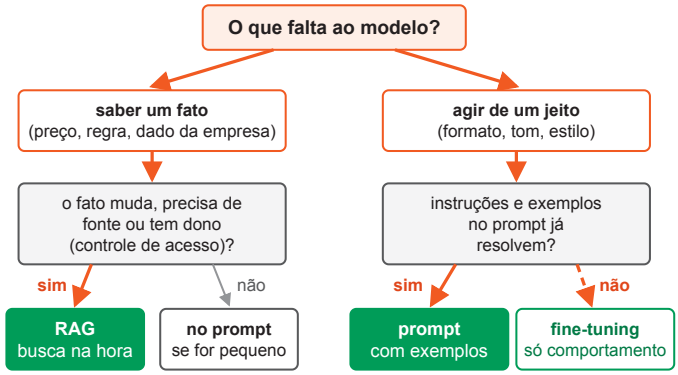
TRÊS JEITOS DE MUDAR UMA RESPOSTA

Pense na resposta do modelo como uma função de três coisas: os pesos θ, o que ele aprendeu no treino; a instrução, o que mandamos fazer; e o contexto, o que ele lê agora:

$$\text{resposta} = \text{LLM}_\theta(\text{instrução},\ \text{contexto})$$

Cada técnica mexe numa peça. A **engenharia de *prompt*** mexe na instrução, sem treino nenhum. O **RAG** (*retrieval-augmented generation*, geração aumentada por recuperação) mexe no contexto: busca na hora os trechos certos e os entrega ao modelo. O ***fine-tuning*** (ajuste fino) mexe nos pesos: treina o modelo de novo com exemplos.

Uma comparação ajuda. Um vendedor novo pode receber uma ordem ("responda em três blocos"), uma tabela impressa na mesa, atualizada toda segunda, ou um curso de uma semana. Ninguém mandaria o vendedor decorar 40 mil preços num curso, se eles mudam na semana seguinte. Mas o jeito de atender, sim, se aprende com treino.



o caminho da seguradora está em laranja: preços pelo RAG, formato pelo prompt e, se ele não bastar, pelo fine-tuning

Figura 1. A primeira pergunta separa o que é informação do que é comportamento. Fato que muda, que precisa de fonte ou que tem dono vai no contexto, pelo RAG. Comportamento começa no prompt e só vai para os pesos, com fine-tuning, quando o prompt não basta.

POR QUE FATO QUE MUDA NÃO VAI NOS PESOS

Há quatro motivos, e cada um sozinho já bastaria.

1. **O modelo aprende mal fato novo por *fine-tuning*.** Ovadia e colegas (2023) compararam as duas formas de injetar conhecimento, e o RAG venceu de forma consistente, tanto para fatos que o modelo já tinha visto quanto para fatos novos. Gekhman e colegas (EMNLP 2024) mostraram que exemplos com conhecimento novo são aprendidos bem mais devagar e que, à medida que o modelo os aprende, a tendência a alucinar cresce de forma linear.
2. **Peso não cita fonte.** O preço decorado vira parte dos números do modelo, sem etiqueta de tabela nem de data.
3. **Peso não tem controle de acesso.** Tudo o que entrou no treino pode sair numa resposta para qualquer usuário. Não existe "esqueça a região Norte para este vendedor".
4. **Atualizar é treinar de novo.** Cada semana vira um ciclo de preparar dados, treinar, avaliar e publicar.

No RAG, a informação fica fora do modelo, numa base versionada, e a busca aplica as regras antes de o modelo ver qualquer coisa:

$$\text{contexto} = \underset{\substack{d \in D \\ \text{região}(d) = r(u) \\ \text{tabela}(d) = \text{vigente}}}{\text{top-k}} \ \text{sim}(q, d)$$

Em palavras: dos itens d da base D, só entram os da região $r(u)$ do usuário e da tabela vigente; entre eles, os k mais parecidos com a pergunta q. A região vem do login, não de um pedido ao modelo.

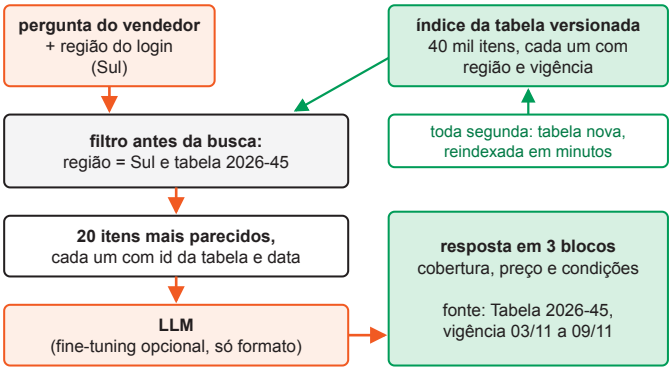


Figura 2. O RAG da seguradora. A região vem do login e filtra a busca antes de qualquer trecho chegar ao modelo. Cada item carrega a tabela e a vigência, então a citação sai dos metadados, e não da memória do modelo. Atualizar preço é reindexar, não treinar.

EXEMPLO RESOLVIDO: O PREÇO DE CADA ATALHO

***Fine-tuning* semanal.** Suponha que preparar os dados, treinar, avaliar e publicar leve $d = 3$ dias. A tabela muda toda segunda, então o modelo novo só entra na quinta. A fração do tempo com preço da semana passada é:

$$\frac{d}{7} = \frac{3}{7} \approx 43\%$$

E isso no melhor caso, se ele tivesse decorado tudo certo. No RAG, a tabela nova é reindexada em minutos.

Tabela inteira no *prompt*. Supondo 30 *tokens* por item (código, cobertura, região, preço e vigência):

$$40.000 \times 30 = 1{,}2 \text{ milhão de tokens por pergunta}$$

São seis janelas de 200 mil. Mesmo num modelo que aceitasse tudo, a US$ 3 por milhão de *tokens* de entrada (preço ilustrativo), cada pergunta custaria US$ 3,60; com 5.000 perguntas por dia, US$ 18 mil por dia. O RAG manda os 20 itens relevantes: $20 \times 30 = 600$ *tokens*.

E ONDE ENTRA O FINE-TUNING?

No formato fixo de três blocos, que é comportamento. A ordem certa é começar pelo mais barato: instrução clara e um exemplo no *prompt*, ou saída estruturada, se a API oferecer. Se, mesmo assim, uma parte das respostas escapa do formato, aí sim um *fine-tuning* leve, como LoRA, com pares de pergunta e resposta no formato certo. Um cuidado: os exemplos de treino não devem ensinar preços. Use valores fictícios marcados ou deixe o preço vir do contexto, para o modelo aprender a forma, e não os números.

EM UMA FRASE

Fato que muda, que precisa de fonte ou que tem dono vai no contexto, pelo RAG; comportamento estável, como formato e tom, começa no *prompt* e vai para os pesos, com *fine-tuning*, só quando o *prompt* não basta.

NÃO CONFUNDA

Fine-tuning ensina **como** responder; RAG entrega **o que** é verdade hoje. Janela de contexto longa não é RAG: ela não filtra por região, não versiona e cobra por cada *token* que você manda.

AGORA, AS ALTERNATIVAS

A *fazer* fine-tuning *semanal do modelo com a tabela nova, para que ele memorize os preços vigentes de cada região.*

Errada. É o atalho que a aula desmonta. O modelo aprende mal fato novo por *fine-tuning* e, ao aprender, alucina mais (Ovadia, 2023; Gekhman, 2024). Mesmo no melhor caso, com três dias de ciclo, ele passaria 43% do tempo com o preço da semana passada. E os pesos não citam a tabela nem escondem de um vendedor os preços das outras regiões.

B *usar RAG sobre a tabela versionada, com filtro por região e citação da fonte, deixando o* fine-tuning*, se preciso, só para o formato.*

Certa. É o desenho da aula. O RAG sobre a tabela versionada traz o preço vigente, o filtro por região aplica o controle de acesso antes de o modelo ler qualquer coisa e a citação sai dos metadados de cada item (tabela e vigência). O *fine-tuning* fica onde ele funciona bem: no comportamento, o formato de três blocos, e só se o *prompt* não bastar.

C *colocar a tabela inteira no prompt de cada requisição, já que os modelos atuais têm janelas de contexto longas.*

Errada. Pela conta da aula, seriam cerca de 1,2 milhão de *tokens* por pergunta, seis janelas de 200 mil, com custo e demora em toda requisição. Os modelos também usam pior o que está no meio de contextos longos (Liu et al., 2024). E o pior: todo vendedor receberia no *prompt* os preços de todas as regiões, o que quebra o controle de acesso.

D *fazer um* fine-tuning *completo uma única vez e pedir, no prompt, que o modelo avise quando um preço puder estar desatualizado.*

Errada. Um modelo treinado uma vez fica desatualizado na semana seguinte. Pedir que ele avise quando um preço "puder estar desatualizado" não resolve: ele não sabe quais preços mudaram, porque não sabe o que não sabe. O aviso viria ao acaso. E continuam faltando a citação e o filtro por região.

E *usar só engenharia de prompt com exemplos de preços, pois o modelo generaliza os valores a partir dos exemplos.*

Errada. Exemplos no *prompt* ensinam forma, não 40 mil preços. Preço não se deduz de exemplos: quando o modelo "generaliza" um valor que não viu, ele inventa um número plausível, que é a definição de alucinação. Sem a tabela no contexto, não há de onde tirar o preço certo.

DE ONDE VEIO A PERGUNTA

"Prompting, RAG or fine-tuning: give me your decision framework with cost and latency attached." Atribuída a OpenAI, Microsoft e Databricks em compilação (Outcome School).

PARA ESTUDAR

- Ovadia et al., Fine-Tuning or Retrieval?, 2023
- Gekhman et al., Does Fine-Tuning LLMs on New Knowledge Encourage Hallucinations?, 2024
- Lewis et al., Retrieval-Augmented Generation, 2020

QUANTAS VOCÊ ACERTOU?

Acertos	Resultado
0 a 3	**Repositor de gôndola.** Você domina o FIFO; os LLMs, ainda não. Tente no próximo processo.
4 a 6	**Encarregado de corredor.** Já desconfia do RAG e sabe onde fica cada produto.
7 a 9	**Gerente de loja.** Pronto para o setor de LLMOps, que fica entre os Frios e o Hortifrúti.
10 a 11	**Engenheiro de IA Sênior.** Superqualificado para repor estoque. O RH vai entrar em contato.

COMO ESTA PROVA FOI FEITA

As 11 questões partem de perguntas que candidatos relataram em entrevistas para engenharia de IA e de *machine learning* em empresas como Google, OpenAI, Anthropic, Microsoft, NVIDIA, Apple, Amazon e Databricks. Quase todas as perguntas reais são abertas. Aqui elas viraram múltipla escolha, e os erros mais comuns de quem responde viraram as alternativas erradas.

Os números dos textos-base vêm das fontes citadas em cada questão. Os relatos vieram de compilações públicas, como a Outcome School, a Aced, a BuildML e a AI Engineering Insider, e de relatos de candidatos publicados na própria Aced. Glassdoor, LeetCode e Reddit bloquearam o acesso automático e não foram usados.

Paródia, sem fins comerciais. Sem vínculo com o Atacadão S.A., o Grupo Carrefour Brasil ou o Inep.