E-BOOK GRATUITO E OPEN SOURCE

Apache Kafka para Entrevistas Java Sênior

Guia prático e gratuito sobre mensageria, arquitetura, reprocessamento, ordenação, idempotência e sistemas distribuídos com Apache Kafka, voltado para desenvolvedores Java Backend em entrevistas técnicas.

João Paulo Rodrigues de Araújo

Versão de 14 de agosto de 2026

CONTRIBUIDORES DESTA VERSÃO

João Paulo Rodrigues de Araújo

Sumário

PARTE I — FUNDAMENTOS

O que é Apache Kafka?

Kafka versus outras ferramentas

PARTE II — ARQUITETURA

Arquitetura interna

Partitions e ordenação

Brokers, líderes e replicação

Consumer Groups e Rebalance

PARTE III — CONSUMO E REPROCESSAMENTO

Offset e Commit

Retention e Replay

Retry e DLQ

PARTE IV — CONFIABILIDADE

Garantias de entrega

Idempotência

Transações e Outbox Pattern

PARTE V — JAVA E SPRING BOOT

Producer com Spring Kafka

Consumer com Spring Kafka

Observabilidade

PARTE I — FUNDAMENTOS

O que é Apache Kafka?

Antes de entrar em Producer, Broker ou Partition, vale resolver uma pergunta que a maioria dos desenvolvedores nunca para para responder com precisão: **por que o Kafka existe?** Se você não consegue justificar a existência de uma peça de infraestrutura, também não consegue justificar quando usá-la — e é exatamente esse tipo de julgamento que uma entrevista sênior tenta medir.

O problema antes do Kafka

Em um sistema com poucos serviços, integração ponto a ponto funciona: o serviço de pagamentos chama diretamente o serviço de notificação via HTTP, que chama o serviço de antifraude, e assim por diante. O problema aparece quando o número de serviços cresce. Cada novo consumidor de um evento de pagamento significa uma nova chamada síncrona, um novo ponto de falha e um novo acoplamento direto entre times.

Imagine um evento de "pagamento aprovado" em um banco digital. Hoje ele pode interessar a: notificação push, antifraude, cashback, contabilidade, analytics e atualização de limite de crédito. Se o serviço de pagamentos chamar cada um desses via HTTP síncrono, ele passa a depender da disponibilidade de todos eles — e a adicionar um sétimo consumidor exige alterar o código do serviço de pagamentos novamente.

> **Não é só sobre performance**
>
> O problema central não é volume de requisições — é **acoplamento**. Um sistema em que o produtor de um evento precisa conhecer todos os seus consumidores não escala organizacionalmente, mesmo que escale em throughput.

Event streaming e arquitetura orientada a eventos

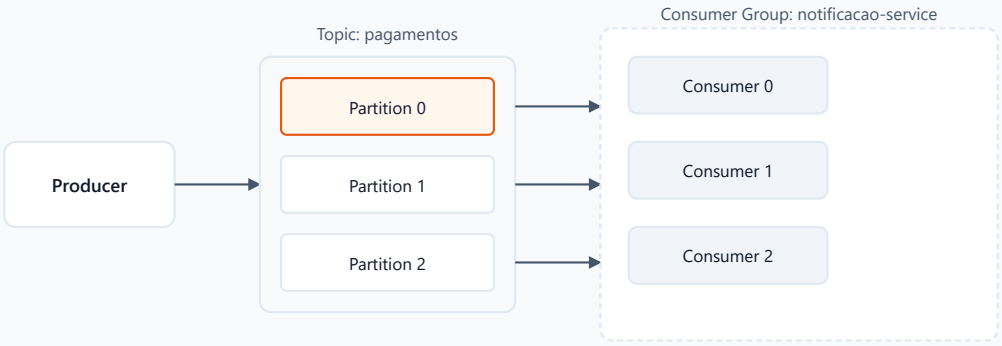
O Kafka nasceu no LinkedIn, por volta de 2010, para resolver o problema de mover grandes volumes de dados de atividade (cliques, visualizações, métricas) entre sistemas de forma confiável e desacoplada. A ideia central é simples de enunciar e profunda na prática: em vez de sistemas se chamando diretamente, eles publicam e consomem **eventos** através de um log distribuído compartilhado.

> **Event streaming**
>
> Event streaming é o padrão arquitetural em que fatos que já aconteceram ("pagamento aprovado", "boleto pago", "cartão bloqueado") são publicados como eventos imutáveis em um log central, e qualquer número de sistemas interessados pode consumi-los de forma independente, no seu próprio ritmo.

Isso é diferente de simplesmente "enviar uma mensagem". Um evento não é um comando ("processe este boleto"); é uma afirmação sobre algo que já ocorreu. O produtor não sabe — e não precisa saber — quem vai consumir o evento nem o que cada consumidor vai fazer com ele. Essa inversão é a base da arquitetura orientada a eventos (*event-driven architecture*): o sistema de pagamentos publica "pagamento aprovado" e segue em frente; antifraude, notificação e contabilidade descobrem esse fato de forma assíncrona, cada um na sua velocidade.

Kafka como log distribuído

Tecnicamente, o Kafka é implementado como um **log distribuído, particionado e replicado**. "Log" aqui não é log de aplicação — é a estrutura de dados: uma sequência ordenada e imutável de registros, cada um identificado por uma posição crescente (o *offset*). Novos eventos são sempre anexados ao final; nada é alterado in-place.



Producer publica no Topic; o Topic é dividido em Partitions; múltiplos Consumers leem de forma independente.

Essa estrutura de log é o que permite duas propriedades que fazem o Kafka ser fundamentalmente diferente de uma fila tradicional:

1. **Múltiplos consumidores independentes** podem ler o mesmo conjunto de eventos, cada um mantendo sua própria posição de leitura (offset), sem que a leitura de um afete a do outro.
2. **Retenção configurável** significa que o evento continua disponível no log depois de consumido — um novo serviço pode nascer amanhã e ler o histórico completo de "pagamento aprovado" dos últimos 7 dias sem que o sistema de pagamentos precise reenviar nada.

Kafka não é apenas uma fila

Essa é provavelmente a comparação mais repetida — e mais incompleta — sobre Kafka.

> **"Kafka é uma fila só que mais rápida"**
>
> Essa afirmação ignora a diferença estrutural mais importante: em uma fila clássica (RabbitMQ, SQS), uma mensagem consumida normalmente **sai da fila** — ela existe para ser entregue uma vez e desaparecer. No Kafka, consumir um evento não o remove do log; ele continua lá até a política de retenção expirá-lo, independentemente de quantos consumidores já o leram.

Uma fila modela **trabalho a ser distribuído** — cada item deve ser processado por exatamente um worker. O Kafka modela **fatos a serem compartilhados** — o mesmo evento pode (e geralmente deve) ser lido por vários sistemas diferentes, cada um com sua própria interpretação do fato. É possível usar Kafka para distribuir trabalho (é isso que um Consumer Group faz dentro de uma partition), mas reduzir o Kafka a isso é perder a metade do motivo pelo qual ele foi desenhado.

Quando utilizar

- Múltiplos sistemas precisam reagir ao mesmo fato de negócio, de forma independente.
- Você precisa de **replay**: reprocessar eventos antigos para corrigir um bug, popular um novo sistema ou reconstruir um estado (ex.: reindexar um Elasticsearch a partir do histórico de eventos).
- O volume e a taxa de eventos são altos o suficiente para que acoplamento síncrono se torne um risco de disponibilidade (um serviço fora do ar não pode travar os demais).
- Você quer desacoplar produtores de consumidores no nível organizacional — times diferentes evoluindo seus serviços sem coordenar deploys.

Quando não utilizar

- Comunicação estritamente request/response, onde quem chama precisa de uma resposta imediata (ex.: autorizar uma transação de cartão em tempo real de ponta a ponta) — isso é papel de uma chamada síncrona (HTTP/gRPC), não de um evento assíncrono.
- Sistemas pequenos, com poucos serviços e baixo volume, onde a complexidade operacional de um cluster Kafka não se paga. Uma fila simples ou até chamadas diretas resolvem com menos esforço.
- Quando a equipe não tem — e não vai construir — a maturidade operacional mínima para lidar com um sistema distribuído com estado (monitoramento de lag, gestão de partições, upgrades de cluster).

Limitações

- Kafka **não garante ordenação global** entre partitions — apenas dentro de uma mesma partition (voltamos a isso no Capítulo 4).
- Não é um banco de dados transacional; embora suporte transações (Capítulo 12), ele não substitui um banco relacional para consultas ad-hoc sobre o estado atual.
- Tem uma curva operacional real: partições mal dimensionadas, rebalances mal compreendidos e retenção mal configurada são causas comuns de incidentes em produção.

Como aparece em entrevistas

Praticamente toda entrevista sobre Kafka começa com uma variação de "o que é Kafka" ou "por que usar Kafka em vez de uma fila". O entrevistador não quer a definição de dicionário — quer ver se você entende o problema de acoplamento que motivou a criação da ferramenta e se você sabe diferenciar "fila de trabalho" de "log de eventos compartilhado". Respostas que citam apenas "alta performance" ou "processa muitos dados" soam decoradas e raramente convencem.

> **Dica de entrevista**
>
> Estruture a resposta em três passos: (1) o problema — acoplamento entre produtores e múltiplos consumidores; (2) a solução — log distribuído e imutável, com múltiplos consumidores independentes; (3) a consequência — replay, desacoplamento organizacional e a diferença fundamental em relação a uma fila.

Relação com Java e Spring Boot

Na prática, um desenvolvedor Java interage com o Kafka através do client oficial (`kafka-clients`) ou, mais comumente, via **Spring Kafka**, que encapsula `KafkaTemplate` para produção e `@KafkaListener` para consumo (Capítulos 13 e 14). O conceito de "publicar um evento e seguir em frente" se traduz diretamente em código: um `@Service` de pagamentos publica um evento via `KafkaTemplate.send(...)` dentro (idealmente) da mesma transação que persiste o pagamento no banco, e retorna, sem esperar por antifraude, notificação ou contabilidade.

Como aparece em sistemas reais

> **PIX recebido, sem Kafka vs. com Kafka**
>
> Sem um log de eventos, o serviço que recebe a confirmação de um PIX precisaria chamar diretamente o serviço de saldo, o de notificação push, o de extrato e o de antifraude — de forma síncrona ou com uma fila dedicada para cada um. Com Kafka, ele publica um único evento `PixRecebido` em um tópico; saldo, notificação, extrato e antifraude consomem esse mesmo evento de forma independente, cada um no seu próprio Consumer Group, sem que o serviço de recebimento do PIX saiba que eles existem.

> **Resumo**
>
> Kafka resolve o problema de acoplamento entre um produtor de eventos e múltiplos consumidores independentes, através de um log distribuído, particionado e replicado, onde eventos são anexados e retidos por um período configurável — não removidos ao serem consumidos. Isso o torna estruturalmente diferente de uma fila tradicional, que modela distribuição de trabalho, não compartilhamento de fatos.

> **Pode vir a seguir**

> Depois de responder "o que é Kafka", prepare-se para: "Kafka é uma fila?", "quando você não usaria Kafka?" e "qual problema real você já resolveu com Kafka?".

PARTE I — FUNDAMENTOS

Kafka versus outras ferramentas

Uma das perguntas mais comuns em entrevista é comparar Kafka com outra ferramenta de mensageria que o candidato já usou. Isso não é bizantinismo de nomenclatura — cada ferramenta foi desenhada para um modelo de uso diferente, e escolher errado tem custo real em produção.

O eixo que realmente importa: fila de trabalho vs. log de eventos

Antes de comparar recursos, é preciso separar as ferramentas em duas famílias:

- **Filas de trabalho** (RabbitMQ, AWS SQS): cada mensagem deve ser processada por exatamente um consumidor; depois de processada (ack), ela normalmente desaparece.
- **Log de eventos / pub-sub com retenção** (Kafka, e — com particularidades — AWS SNS + Google Pub/Sub): o mesmo evento pode ser lido por múltiplos consumidores independentes, e pode continuar disponível depois de lido.

Ignorar essa diferença é a causa mais comum de escolher a ferramenta errada.

Kafka vs. RabbitMQ

Kafka vs. RabbitMQ

Critério	Kafka	RabbitMQ
Modelo	Log distribuído, particionado	Fila de mensagens (com exchanges/routing)
Retenção	Configurável (tempo/tamanho), mensagem persiste após consumo	Mensagem some após ack (a menos que configurado de outra forma)
Replay	Nativo — resetar offset relê o histórico	Não nativo — exige reenvio manual
Ordenação	Garantida por partition	Garantida por fila (FIFO), mais simples de raciocinar em fila única
Múltiplos consumidores do mesmo dado	Natural, via Consumer Groups distintos	Exige fan-out explícito (exchange do tipo *fanout*)
Throughput em alta escala	Muito alto, desenhado para volume	Alto, mas com overhead maior de roteamento por mensagem
Roteamento complexo (regras, prioridade)	Limitado — roteamento é por tópico/partition/key	Rico — exchanges, routing keys, prioridade de mensagem
Curva operacional	Mais alta (cluster, partições, retenção)	Mais baixa para casos simples

RabbitMQ brilha quando o requisito é roteamento sofisticado de mensagens de trabalho — prioridades, dead lettering fino, múltiplos padrões de exchange. Kafka brilha quando o requisito é um histórico de eventos compartilhado por múltiplos consumidores, com necessidade de replay.

Kafka vs. AWS SQS

Kafka vs. AWS SQS

Critério	Kafka	AWS SQS
Modelo	Log distribuído	Fila gerenciada (Standard ou FIFO)
Retenção após consumo	Sim (até o limite de retenção do tópico)	Não — mensagem é removida após ack (ou expira)
Replay	Nativo	Não — só via DLQ redrive (reprocessar mensagens que falharam, não histórico completo)
Ordenação	Por partition	FIFO garante ordem por *message group*; Standard não garante ordem
Operação	Você gerencia o cluster (ou usa serviço gerenciado como MSK/Confluent Cloud)	Totalmente gerenciado pela AWS, zero infraestrutura
Escala de consumidores independentes	Nativa (múltiplos Consumer Groups)	Requer SNS + múltiplas filas para fan-out
Throughput	Muito alto	Alto, com limites de throughput por fila (Standard escala melhor que FIFO)

SQS FIFO não é Kafka

SQS FIFO garante ordem dentro de um *message group* e evita duplicidade — mas continua sendo uma fila: a mensagem lida some da fila. Ele resolve ordenação, não

> resolve retenção/replay nem múltiplos consumidores independentes do mesmo dado sem duplicar a mensagem via SNS.

Kafka vs. AWS SNS

SNS é um serviço de **pub/sub puro** (fan-out): uma mensagem publicada é entregue a todos os assinantes (filas SQS, endpoints HTTP, Lambda). É comum encontrar o padrão **SNS + SQS**: SNS distribui, cada SQS consome de forma independente — isso reproduz parcialmente o que Kafka faz nativamente com Consumer Groups, mas sem retenção de longo prazo nem replay do histórico: uma vez entregue e processada, a mensagem não pode ser relida a partir do SNS.

Kafka vs. Google Pub/Sub

Kafka vs. Google Pub/Sub

Critério	Kafka	Google Pub/Sub
Modelo	Log particionado, gerenciado por você (ou serviço gerenciado)	Pub/sub totalmente gerenciado
Retenção	Configurável, tipicamente dias	Configurável, até 31 dias (mensagens já confirmadas por padrão não ficam retidas do mesmo jeito)
Replay	Nativo via offset	Via *seek*, suportado, mas com semântica diferente de offset por partition
Ordenação	Por partition (key)	Por *ordering key*, opcional e com custo de throughput
Operação	Requer gestão de cluster (ou serviço gerenciado)	Zero infraestrutura

Pub/Sub do Google é operacionalmente o mais próximo de "Kafka sem operar Kafka", mas o modelo de partições e o ecossistema de client (especialmente para Java/Spring) ainda favorecem o Kafka em times que já têm essa stack.

Critério de decisão prático

> **Dica de entrevista**
>
> Quando pedirem para comparar, não decore a tabela — explique o critério de decisão: "se preciso que múltiplos serviços independentes leiam o mesmo evento, com possibilidade de replay, penso em Kafka; se preciso distribuir trabalho entre

workers, com roteamento rico e baixa necessidade de retenção, penso em RabbitMQ ou SQS". Isso demonstra raciocínio, não memorização.

Escolhendo a ferramenta certa para cada parte do fluxo de pagamento

Em um sistema de pagamentos real, é comum usar **as duas famílias ao mesmo tempo**: Kafka para o evento de domínio "pagamento aprovado" (que antifraude, contabilidade e notificação consomem de forma independente e podem precisar reprocessar), e uma fila SQS para o trabalho interno de "enviar e-mail de confirmação" (uma tarefa que deve ser processada exatamente uma vez por um worker, sem necessidade de replay).

Resumo

RabbitMQ e SQS modelam filas de trabalho — mensagem processada some. Kafka modela um log de eventos retido, lido por múltiplos consumidores independentes, com replay nativo. SNS resolve fan-out, mas não retenção de longo prazo. Pub/Sub do Google é o mais próximo operacionalmente do Kafka "sem operar cluster", mas com semântica de ordenação e retenção diferente. A escolha certa depende do que se precisa fazer com a mensagem depois que ela é lida a primeira vez.

Pode vir a seguir

Prepare-se também para: "você usaria Kafka para tudo?", "como você faria fan-out sem Kafka?" e "o que aconteceria se você usasse SQS para um caso que precisa de replay?".

PARTE II — ARQUITETURA

Arquitetura interna

Este capítulo define, de forma precisa, as peças que vão aparecer em praticamente todo o resto do livro. Se o Capítulo 1 respondeu "por que Kafka existe", este capítulo responde "do que ele é feito".

Producer

> **Producer**
>
> Producer é qualquer aplicação que publica eventos em um tópico Kafka. Na prática, é um microsserviço (ou parte de um) que, ao final de uma operação de negócio, envia uma mensagem representando o fato que acabou de ocorrer.

Um Producer decide para qual **tópico** enviar, qual **key** usar (Capítulo 4) e como serializar o valor (JSON, Avro, Protobuf). Ele não sabe — nem deveria saber — quantos consumidores existem ou o que fazem com o evento. Em Spring Boot, um Producer é tipicamente um `@Service` usando `KafkaTemplate<K, V>` (Capítulo 13).

Broker

> **Broker**
>
> Broker é um processo Kafka em execução, responsável por armazenar dados de um subconjunto das partitions do cluster e servir requisições de leitura e escrita para essas partitions.

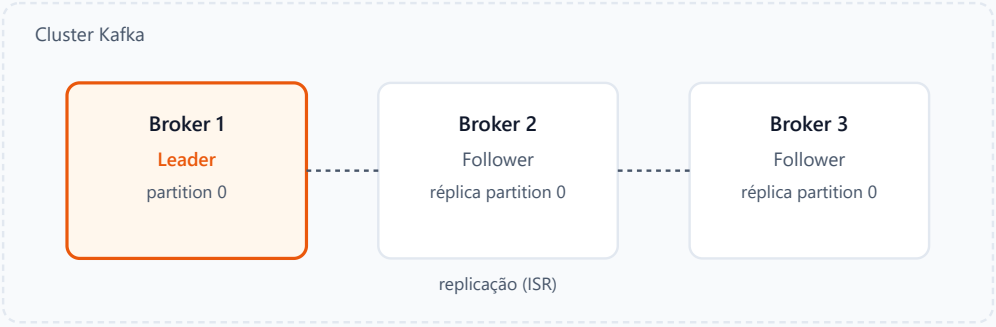
Um broker sozinho já é um Kafka funcional, mas em produção ele sempre roda em conjunto com outros brokers, formando um **cluster**, para tolerância a

falhas e distribuição de carga.

Cluster

Cluster

Cluster é o conjunto de brokers que, juntos, armazenam e servem todos os tópicos de uma instalação Kafka. A coordenação entre brokers (eleição de controlador, metadados do cluster) é feita via Kafka Raft (KRaft) nas versões atuais, substituindo o antigo ZooKeeper.



Um cluster com três brokers replicando as partitions de um tópico entre si.

Topic

Topic

Topic é o nome lógico sob o qual eventos relacionados são publicados e consumidos — por exemplo, `pagamentos.aprovados` ou `pix.recebido`. Um tópico é uma abstração; fisicamente, ele é dividido em uma ou mais partitions.

Um tópico não tem schema imposto pelo Kafka em si (o schema, quando existe, é responsabilidade de uma camada como o Schema Registry, fora do escopo deste capítulo). O que o Kafka garante é a ordem de entrega **dentro** de cada partition do tópico — não entre tópicos diferentes, nem necessariamente entre partitions do mesmo tópico.

Partition

> **Partition**
>
> Partition é uma subdivisão ordenada e imutável de um tópico — o log físico propriamente dito. Cada partition é uma sequência de registros identificados por um offset crescente, armazenada em um broker (com réplicas em outros brokers, ver Capítulo 5).

Partitions existem por dois motivos: **paralelismo** (múltiplos consumidores podem processar partitions diferentes ao mesmo tempo) e **escalabilidade horizontal** (o tópico pode crescer distribuindo partitions entre mais brokers). O Capítulo 4 aprofunda como a escolha da key determina em qual partition uma mensagem cai, e por que isso é a decisão de design mais importante ao modelar um tópico.

Offset

> **Offset**
>
> Offset é a posição sequencial de um registro dentro de uma partition — um número inteiro crescente, único por partition, que identifica univocamente uma mensagem naquela partition.

Offset não é global ao tópico; é local a cada partition. O mesmo número de offset existe de forma independente em cada partition de um tópico. O Capítulo 7 detalha como consumidores usam o offset para saber "até onde já li".

Consumer

> **Consumer**
>
> Consumer é qualquer aplicação que lê eventos de um ou mais tópicos, processando-os a partir da posição (offset) em que parou da última vez.

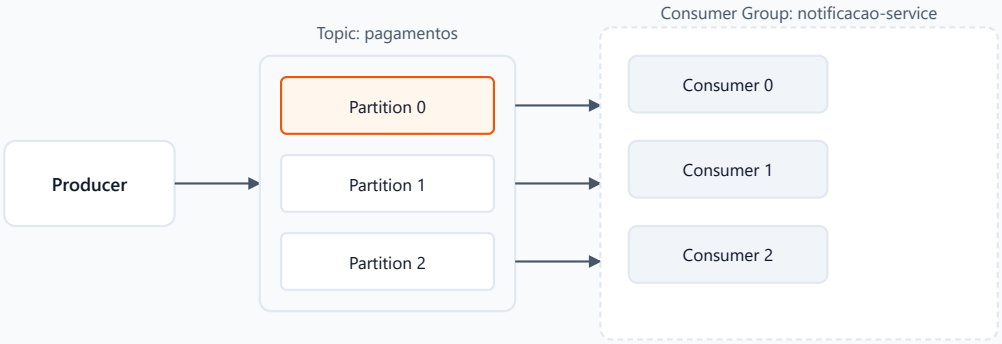
Um Consumer, sozinho, lê de todas as partitions às quais está atribuído. Em Spring Boot, o padrão é um método anotado com `@KafkaListener` (Capítulo 14).

Consumer Group

> **Consumer Group**
>
> Consumer Group é um conjunto de instâncias de consumer que se identificam com o mesmo `group.id` e dividem entre si as partitions de um tópico — cada partition é lida por exatamente um consumer do grupo em um dado momento.

É o Consumer Group que transforma o Kafka em uma ferramenta capaz de **tanto** distribuir trabalho (dentro de um grupo, cada partition vai para um único consumer, como em uma fila) **quanto** compartilhar o mesmo dado com múltiplos sistemas (grupos diferentes leem o mesmo tópico de forma totalmente independente, cada um com seu próprio conjunto de offsets). O Capítulo 6 detalha o comportamento de rebalance quando consumers entram ou saem de um grupo.

Juntando as peças



Producer publica no Topic pagamentos, dividido em 3 partitions; o Consumer Group notificacao-service tem um consumer por partition.

O fluxo completo: um Producer publica no tópico `pagamentos`; o Kafka decide (via key/hash, Capítulo 4) em qual das partitions do tópico a mensagem cai; ela é persistida com um offset sequencial naquela partition, replicada para os brokers de réplica (Capítulo 5); um ou mais Consumer Groups leem a partition de forma independente, cada grupo avançando seu próprio conjunto de offsets.

Quando esse modelo é a escolha certa

Esse desenho compensa quando o número de eventos e de consumidores independentes justifica a complexidade — tipicamente a partir de alguns microsserviços reagindo ao mesmo domínio de eventos. Para um único produtor/consumidor com baixo volume, a mesma arquitetura tecnicamente funciona, mas o custo operacional de manter partitions, réplicas e um cluster raramente se paga.

Limitações a ter em mente

- O número de partitions de um tópico define o paralelismo máximo de consumo dentro de um Consumer Group — aumentar depois é possível, mas redistribui as chaves (Capítulo 4).
- Mais partitions não é sempre melhor: cada partition consome recursos de broker (file handles, memória) e aumenta o tempo de rebalance.

Como aparece em entrevistas

Entrevistadores costumam pedir para desenhar esse fluxo no quadro (ou verbalmente) e depois variar os parâmetros: "o que muda se eu tiver 2 partitions e 5 consumers no grupo?", "e se eu tiver dois Consumer Groups diferentes lendo o mesmo tópico?". Responder com precisão sobre offset ser local à partition — não ao tópico — é um sinal frequentemente usado para diferenciar quem só usou Kafka de quem entende Kafka.

> **Dica de entrevista**
>
> Ao descrever a arquitetura, deixe explícito que offset é por partition, não por tópico, e que Consumer Groups diferentes são completamente independentes entre si — dois grupos lendo o mesmo tópico não competem pelas mesmas mensagens.

Relação com Java e Spring Boot

Cada peça mapeia para uma configuração ou classe do Spring Kafka: `bootstrap-servers` aponta para os brokers do cluster; `KafkaTemplate.send(topic, key, value)` é o Producer; `@KafkaListener(topics = "...", groupId = "...")` define o Consumer e seu Consumer Group; o offset é gerenciado pelo `Acknowledgment` (commit manual) ou automaticamente pelo client, conforme configuração (Capítulo 7).

Faturas de cartão

O tópico `faturas.fechadas` pode ter 12 partitions, uma estratégia de key por `contaId` (para garantir que todos os eventos da mesma conta fiquem ordenados na mesma partition) e dois Consumer Groups: um do serviço de cobrança (que gera o boleto) e outro do serviço de analytics (que alimenta um dashboard) — ambos leem o mesmo tópico, de forma totalmente independente, cada um na sua velocidade.

Resumo

Producer publica, Broker armazena, Cluster é o conjunto de brokers, Topic é o nome lógico, Partition é o log físico ordenado, Offset é a posição dentro da partition, Consumer lê, e Consumer Group é o mecanismo que permite tanto dividir trabalho quanto compartilhar o mesmo dado entre sistemas independentes.

Pode vir a seguir

A seguir, é comum perguntarem: "por que existem partitions?", "como o Kafka escolhe a partition de uma mensagem?" e "o que acontece se eu tiver mais consumers do que partitions?".

PARTE II — ARQUITETURA

Partitions e ordenação

O Capítulo 3 definiu partition como "o log físico propriamente dito". Este capítulo explica por que essa peça específica existe, como ela transforma a escolha de uma key em uma decisão de arquitetura, e por que "o Kafka garante ordenação" é uma frase incompleta até você dizer *dentro de quê*.

Por que partitions existem

Um tópico poderia, em princípio, ser um único log gigante. Na prática isso não escalaria: um log é lido e escrito sequencialmente por um processo de broker, e um Consumer Group só consegue paralelizar o consumo de um tópico até o limite do número de partitions que ele tem. Partitions existem para resolver dois problemas ao mesmo tempo:

> **Por que partitions existem**
>
> Partitions dividem um tópico em unidades independentes de armazenamento e consumo, permitindo que o Kafka distribua a carga de escrita entre múltiplos brokers (escalabilidade) e que múltiplos consumers de um mesmo Consumer Group processem partitions diferentes em paralelo (paralelismo).

Sem partitions, adicionar mais consumers a um Consumer Group não aumentaria o throughput de consumo — só existiria um único fluxo sequencial para processar. Com partitions, o paralelismo de consumo escala junto com o número de partitions do tópico.

Paralelismo e escalabilidade

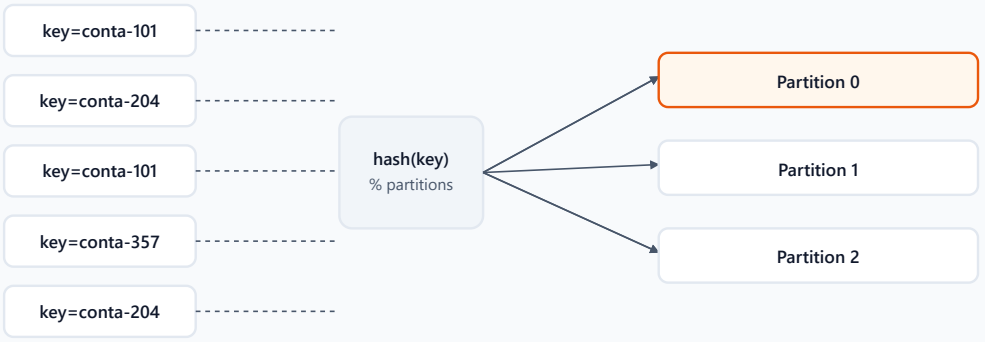
Os dois motivos não são a mesma coisa, e vale separá-los:

- **Paralelismo de consumo**: dentro de um Consumer Group, cada partition é atribuída a exatamente um consumer por vez. Mais partitions significa mais consumers processando ao mesmo tempo — até o limite de uma partition por consumer (Capítulo 6 aprofunda o que acontece quando esse limite é ultrapassado).
- **Escalabilidade de armazenamento e escrita**: partitions de um mesmo tópico podem residir em brokers diferentes. Um tópico com 12 partitions distribuídas em 3 brokers tem 3 vezes mais capacidade de escrita simultânea do que se estivesse inteiro em um único broker.

Message key e a decisão de particionamento

Quando um Producer publica uma mensagem, ele pode (opcionalmente) informar uma **key**. O broker usa essa key para decidir em qual partition a mensagem cai — e essa decisão é determinística: a mesma key sempre resulta na mesma partition, para um número fixo de partitions no tópico.

> **Key e hash**
>
> O particionador padrão do Kafka calcula `hash(key) % número_de_partitions` para decidir a partition de destino. Como o hash de uma mesma key é sempre o mesmo, todas as mensagens com essa key caem consistentemente na mesma partition — e, portanto, são lidas na ordem em que foram escritas.



Mensagens com a mesma key (conta-101, conta-204) são roteadas por hash sempre para a mesma partition.

Quando a mensagem **não** tem key, o particionador distribui as mensagens entre as partitions de forma aproximadamente uniforme (as versões recentes do client Java usam uma estratégia *sticky*, que agrupa lotes de mensagens em uma partition por vez antes de rotacionar, otimizando o batching sem comprometer a distribuição). O preço dessa distribuição é que não existe garantia de ordem alguma entre essas mensagens.

Ordenação dentro da partition — e a ausência de ordenação global

> **Ordenação no Kafka**
>
> O Kafka garante que mensagens escritas em uma mesma partition são lidas na mesma ordem em que foram escritas. Ele não garante nenhuma ordem entre mensagens de partitions diferentes do mesmo tópico.

Isso decorre diretamente da estrutura de log: uma partition é uma sequência linear e imutável, então a ordem de escrita é, por construção, a ordem de leitura. Mas um tópico com múltiplas partitions não é uma única sequência — é um conjunto de sequências independentes, cada uma com seu próprio

relógio de offsets. Duas mensagens em partitions diferentes não têm relação de ordem definida entre si, mesmo que uma tenha sido escrita visivelmente antes da outra em tempo real.

> **"Ordenado" não é uma propriedade do tópico — é uma propriedade da partition**
>
> É comum ouvir "esse tópico é ordenado" como se fosse uma propriedade binária do tópico inteiro. A frase correta é "os eventos desta key estão ordenados entre si", porque só isso é garantido — e só é garantido porque a key fixa a mensagem em uma única partition.

Escolhendo a key: a decisão mais importante ao modelar um tópico

A pergunta a se fazer não é "qual key distribui melhor as partitions", mas **"quais eventos precisam estar ordenados entre si?"**. A resposta a essa pergunta é a key.

- Se dois eventos do mesmo `contaId` precisam ser processados na ordem em que ocorreram (ex.: "PIX recebido" seguido de "saldo debitado por estorno"), a key deve ser `contaId` — não `eventId`, que geraria melhor distribuição, mas destruiria a ordenação entre esses dois eventos.
- Se os eventos são independentes entre si (não existe relação de ordem que importe), otimizar a distribuição com uma key de alta cardinalidade (ou nenhuma key) é uma escolha válida e melhora o balanceamento de carga entre partitions.

> **"Sempre uso um UUID aleatório como key para distribuir bem"**
>
> Um UUID aleatório por mensagem maximiza a distribuição, mas também garante que mensagens relacionadas (do mesmo pedido, da mesma conta, do mesmo contrato) caiam em partitions diferentes — eliminando qualquer garantia de ordem entre elas. Se ordenação relativa importa para o domínio, isso é um bug de modelagem, não uma otimização.

Aumentar o número de partitions: o que muda

Aumentar as partitions de um tópico existente é possível, mas tem uma consequência que passa despercebida em muitas entrevistas: a fórmula `hash(key) % número_de_partitions` muda de resultado quando o denominador muda. Mensagens novas com a mesma key de antes podem cair em uma partition diferente das mensagens antigas — a ordenação histórica daquela key, antes preservada em uma única partition, passa a ficar dividida entre duas.

> **Aumentar partitions não é uma operação neutra**
>
> Depois de aumentar o número de partitions, novas mensagens de uma key já existente podem ir para uma partition diferente da que continha o histórico dessa key. Isso não corrompe dados, mas quebra a suposição de que "toda a história de uma key está em uma única partition" — algo que consumers com lógica dependente de ordem histórica precisam saber lidar (ou o time precisa planejar o aumento de partitions com essa consequência em mente, tipicamente migrando para um tópico novo em vez de só aumentar o existente).

Reduzir o número de partitions de um tópico, por outro lado, não é suportado nativamente pelo Kafka — a única forma é recriar o tópico.

Impactos arquiteturais da escolha de partitions

- **Poucas partitions** limitam o paralelismo máximo de consumo do Consumer Group e concentram carga de escrita em poucos brokers.
- **Partitions demais** aumentam o overhead de metadados do cluster, o tempo de rebalance (Capítulo 6) e o número de file handles abertos por broker — sem benefício adicional se não há consumers suficientes para aproveitar o paralelismo extra.
- O número de partitions deve ser dimensionado pelo throughput-alvo e pelo número máximo de consumers que o Consumer Group pretende escalar, não por um valor arbitrário "redondo".

Como aparece em entrevistas

Depois de "o que é uma partition", a pergunta natural é sobre a key: "como o Kafka decide em qual partition uma mensagem cai?" e, em seguida, a pegadinha clássica: "o Kafka garante ordenação?" — que precisa ser respondida com a ressalva de partition, nunca como um "sim" ou "não" isolado.

> **Dica de entrevista**
>
> Sempre que disser "o Kafka garante ordenação", complete a frase com "dentro da partition, determinada pela key". Entrevistadores sênior costumam repetir a pergunta de formas diferentes só para checar se essa ressalva aparece de forma consistente, não decorada uma única vez.

Relação com Java e Spring Boot

No Spring Kafka, a key é o segundo argumento de `KafkaTemplate.send(topic, key, value)`. Quando nenhuma key é necessária, o overload de dois argumentos (`send(topic, value)`) publica sem key, sujeito à distribuição aproximadamente uniforme descrita acima. É comum ver times esquecerem de passar a key em um refactor e só perceberem a perda de ordenação em produção, quando processamento fora de ordem começa a gerar inconsistência de estado — daí a importância de tratar a key como parte do contrato do tópico, não como um detalhe de implementação do producer.

> **Fatura de cartão: por que a key é o contaId**
>
> No tópico `faturas.eventos` , eventos como `FaturaFechada` , `PagamentoRegistrado` e `FaturaReaberta` da mesma conta precisam ser processados na ordem em que ocorreram — processar `PagamentoRegistrado` antes de `FaturaFechada` levaria o serviço de cobrança a uma conclusão errada

sobre o saldo devedor. Usar `contaId` como key garante que esses três eventos, para uma mesma conta, fiquem na mesma partition e sejam lidos na ordem certa — mesmo que contas diferentes sejam processadas em paralelo, em partitions diferentes.

Resumo

Partitions existem para paralelizar consumo e distribuir escrita entre brokers. A key determina, via hash, em qual partition uma mensagem é armazenada — e é essa escolha, não o Kafka em si, que decide quais eventos ficam ordenados entre si. Ordenação é garantida dentro da partition; não existe ordenação global entre partitions de um tópico. Aumentar o número de partitions muda o resultado do hash para novas mensagens, quebrando a suposição de que o histórico de uma key está inteiro em uma única partition.

Pode vir a seguir

Prováveis follow-ups: "como você escolheria a key para X sistema?", "o que acontece se eu tiver mais consumers do que partitions?" e "como você lidaria com uma partition muito maior que as outras (partition hot spot) por causa de uma key de alta frequência?".

PARTE II — ARQUITETURA

Brokers, líderes e replicação

Os Capítulos 3 e 4 trataram partition como uma unidade lógica de ordenação e paralelismo. Este capítulo olha para a mesma partition do ponto de vista de disponibilidade: onde ela vive fisicamente, quantas cópias dela existem, e o que acontece quando um broker desaparece do cluster sem aviso.

O problema que a replicação resolve

Uma partition, sozinha, é um arquivo de log em um único broker. Se esse broker cair — disco corrompido, falha de hardware, atualização mal feita, zona de disponibilidade fora do ar — a partition inteira fica inacessível: nenhum producer consegue escrever nela, nenhum consumer consegue lê-la, até o broker voltar (se voltar). Para um sistema que se propõe a ser a espinha dorsal de comunicação entre microsserviços financeiros, essa é uma dependência inaceitável.

> **Replication Factor**
>
> Replication Factor é o número de cópias de cada partition mantidas em brokers diferentes do cluster. Um replication factor de 3 significa que cada partition do tópico tem 3 réplicas — uma delas o leader, as outras duas followers — espalhadas por brokers distintos.

Leader e Follower

> **Leader**

> O leader de uma partition é o broker que serve todas as leituras e escritas dela. Producers e consumers só interagem com o leader — nunca diretamente com um follower (na configuração padrão).

> **Follower**
>
> Um follower replica continuamente os dados do leader, mantendo uma cópia atualizada da partition, mas sem servir leituras ou escritas de clientes. Sua única função é estar pronto para assumir como leader se o atual falhar.

Cada partition tem exatamente um leader e zero ou mais followers, dependendo do replication factor. Um cluster com replication factor 3 distribui essas 3 réplicas em brokers diferentes — nunca duas réplicas da mesma partition no mesmo broker, o que anularia a proteção contra a perda desse broker.

ISR: quem pode virar leader

Nem todo follower está necessariamente pronto para assumir a liderança a qualquer momento. Um follower pode estar temporariamente atrasado — uma rede lenta, uma pausa de I/O — e permitir que ele vire leader nesse estado significaria aceitar uma versão desatualizada da partition como se fosse a fonte da verdade.

> **ISR (In-Sync Replicas)**
>
> ISR é o subconjunto de réplicas — incluindo o próprio leader — que estão suficientemente atualizadas com os dados confirmados. Apenas réplicas no ISR são elegíveis para eleição como novo leader quando o atual falha.

Um follower que fica para trás (por exemplo, por uma falha de rede prolongada) é removido do ISR até se atualizar novamente. Isso é o que

impede uma réplica desatualizada de virar leader e "esquecer" mensagens já confirmadas a producers.

O que acontece quando o Leader cai

Antes: Broker 1 é leader da partition. Depois de sua queda, o Broker 2 (follower no ISR) é eleito novo leader.

Quando o broker que hospeda o leader de uma partition fica indisponível, o controlador do cluster (eleito via KRaft) detecta a falta de heartbeat e promove um follower do ISR a novo leader — automaticamente, sem intervenção manual. Producers e consumers que estavam apontando para o leader antigo recebem um erro transitório do client, que automaticamente redescobre o novo leader e retoma a operação. Do ponto de vista da aplicação, isso se manifesta como uma pequena latência elevada durante a transição, não como uma falha visível — desde que o `acks` do producer e a configuração de retry do client estejam corretos (Capítulo 10).

> **Failover automático não é o mesmo que zero impacto**
>
> A eleição de um novo leader é automática, mas não é instantânea nem gratuita: durante a janela de detecção e eleição (tipicamente segundos, não minutos, em um

> cluster saudável), escritas e leituras daquela partition específica ficam temporariamente indisponíveis. Sistemas sensíveis a latência precisam considerar esse intervalo no seu orçamento de disponibilidade, não assumir failover como algo transparente e instantâneo.

Distribuição das réplicas

O Kafka tenta distribuir as réplicas de cada partition entre brokers diferentes de forma balanceada — tanto para espalhar carga de leitura/escrita quanto para reduzir a chance de que uma única falha de infraestrutura (um rack, uma zona de disponibilidade) derrube múltiplas réplicas da mesma partition ao mesmo tempo. Em clusters que rodam em múltiplas zonas de disponibilidade de nuvem, é possível configurar *rack awareness* para garantir que réplicas de uma mesma partition fiquem em zonas diferentes — sem isso, um replication factor de 3 ainda protege contra falha de um broker, mas não necessariamente contra a queda de uma zona inteira, se todas as réplicas calharem de estar concentradas nela.

Quando aumentar o Replication Factor

Replication Factor: trade-offs

Replication Factor	Tolerância a falhas	Custo
1	Nenhuma — perda do broker é perda da partition	Mínimo (sem overhead de replicação)
2	Tolera a perda de 1 broker, mas com margem apertada durante a recuperação	Moderado
3 (padrão recomendado em produção)	Tolera a perda de 1 broker com folga, ou 2 falhas não simultâneas	Maior uso de disco e rede

Replication factor 1 é aceitável apenas em ambiente de desenvolvimento local. Em produção, 3 é o padrão de mercado — é o menor número que permite tolerar a perda de um broker e ainda manter mais de uma réplica no ISR durante a recuperação.

Limitações

- Replicação protege contra falha de broker, não contra corrupção de dados na origem: se o producer grava um valor errado, a replicação propaga esse valor errado com a mesma confiabilidade.
- Mais réplicas significam mais uso de disco, rede e I/O — replication factor não é uma configuração "quanto maior, melhor" sem custo.
- Replicação é sobre disponibilidade da partition, não sobre backup de longo prazo ou disaster recovery entre regiões geográficas distintas, que exigem estratégias adicionais (ex.: MirrorMaker, replicação entre clusters).

Como aparece em entrevistas

Depois de explicar Leader/Follower, o entrevistador tipicamente pergunta "o que acontece se o broker leader cair?" — testando se você sabe que a eleição é automática, vem do ISR, e tem uma janela de indisponibilidade real, não instantânea. Uma resposta comum e incompleta é "o Kafka elege outro leader automaticamente", sem mencionar o ISR nem o impacto transitório.

> **Dica de entrevista**
>
> Ao responder sobre queda de broker, cite explicitamente que o novo leader vem do ISR (não de qualquer follower) e que existe uma janela de indisponibilidade durante a detecção e eleição — isso mostra entendimento do mecanismo, não só do resultado final.

Relação com Java e Spring Boot

Do ponto de vista do código Spring Kafka, replicação e eleição de leader são inteiramente transparentes: o client Kafka embutido no Spring Boot descobre o leader atual de cada partition via metadados do cluster e redireciona requisições automaticamente após uma eleição. O que o desenvolvedor Java configura é `acks` no producer (Capítulo 10) e o `min.insync.replicas` do tópico — que define quantas réplicas no ISR precisam confirmar uma escrita para ela ser considerada bem-sucedida quando `acks=all`.

> **Tópico de transações de cartão**
>
> Um tópico `cartao.transacoes.autorizadas` roda com replication factor 3 e `min.insync.replicas=2`. Isso significa que toda escrita com `acks=all` só é confirmada ao producer depois que pelo menos 2 das 3 réplicas (leader incluído) a persistirem — garantindo que, mesmo se o broker leader cair logo em seguida, a

transação já está segura em pelo menos uma réplica que pode assumir a liderança sem perda de dados.

Resumo

Replication Factor define quantas cópias de cada partition existem, distribuídas entre brokers diferentes. O leader serve todas as leituras e escritas; followers replicam passivamente. O ISR é o subconjunto de réplicas atualizadas o suficiente para serem elegíveis como novo leader. Quando o leader cai, o controlador do cluster elege automaticamente um novo leader a partir do ISR — um processo automático, mas não instantâneo. Replication factor 3 é o padrão de produção, equilibrando tolerância a falhas e custo de infraestrutura.

Pode vir a seguir

Prováveis follow-ups: "o que é `min.insync.replicas` e como ele se relaciona com `acks=all`?", "o que acontece se todas as réplicas do ISR caírem ao mesmo tempo?" e "qual a diferença entre Partition e Replication Factor?".

PARTE II — ARQUITETURA

Consumer Groups e Rebalance

O Capítulo 3 definiu Consumer Group como "o mecanismo que permite tanto dividir trabalho quanto compartilhar o mesmo dado entre sistemas independentes". Este capítulo abre essa definição: como a divisão de partitions funciona na prática, o que acontece quando o número de consumers muda, e por que rebalance é um dos tópicos mais mal compreendidos do Kafka em produção.

Como um Consumer Group divide o trabalho

> **Consumer Group**
>
> Um Consumer Group é um conjunto de instâncias de consumer identificadas pelo mesmo `group.id`. O Kafka garante que, a qualquer momento, cada partition de um tópico assinado é atribuída a exatamente um consumer do grupo — nunca dois consumers do mesmo grupo lendo a mesma partition simultaneamente.

Essa regra — uma partition, um consumer, por grupo — é o que torna o paralelismo de consumo previsível: se um tópico tem 6 partitions e o grupo tem 3 consumers, cada consumer processa, em média, 2 partitions. Isso também define o limite superior de paralelismo: não existe forma de um Consumer Group processar um tópico com mais paralelismo do que o número de partitions permite.

O que acontece com números diferentes de consumers

Partitions vs. consumers no mesmo Consumer Group

Cenário	O que acontece
Partitions = Consumers	Cada consumer processa exatamente 1 partition — paralelismo máximo alcançado
Partitions > Consumers	Alguns consumers processam mais de uma partition — ainda paralelo, mas desbalanceado
Consumers > Partitions	Os consumers excedentes ficam ociosos — nenhuma partition sobra para atribuir a eles

Consumers ociosos não são um bug — são um limite conhecido

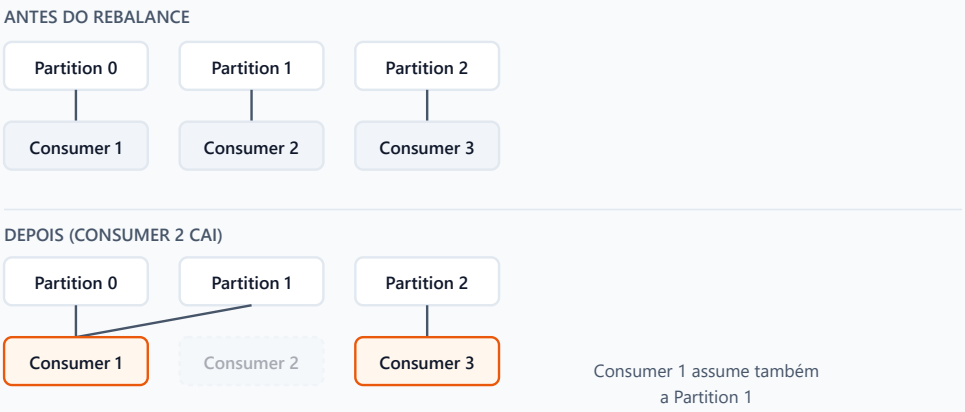
Se um tópico tem 4 partitions e o Consumer Group escala para 6 instâncias, 2 delas simplesmente não recebem nenhuma partition e ficam ociosas, sem processar nada, até que uma das outras 4 caia ou o número de partitions aumente. Isso costuma surpreender times que escalam consumers via Kubernetes esperando ganho de throughput proporcional, sem considerar o número de partitions do tópico como teto.

O que dispara um rebalance

Rebalance

Rebalance é o processo de redistribuir as partitions de um tópico entre os consumers ativos de um grupo, disparado sempre que a composição do grupo muda: um consumer entra, um consumer sai voluntariamente, ou um consumer é

> declarado morto por não enviar heartbeat dentro do tempo configurado (`session.timeout.ms`).



Antes: 3 consumers, 1 partition cada. O Consumer 2 cai; depois do rebalance, Consumer 1 e Consumer 3 dividem as 3 partitions entre si.

O gatilho mais comum em produção não é uma queda abrupta de processo — é um consumer que demora demais para processar um lote de mensagens (`max.poll.interval.ms` excedido) e é considerado morto pelo grupo mesmo continuando vivo, só lento. Esse cenário gera um rebalance desnecessário que expulsa um consumer saudável, e é uma causa clássica de "rebalances infinitos" em produção quando o processamento por mensagem é consistentemente mais lento que o intervalo configurado.

O impacto de um rebalance

Na estratégia clássica ("eager rebalance"), todos os consumers do grupo revogam **todas** as suas partitions no início do rebalance, mesmo aquelas que continuarão com o mesmo consumer depois — o grupo inteiro para de processar durante a janela de reatribuição. Em grupos grandes ou com processamento pesado por partition, essa pausa pode ser perceptível.

Cooperative rebalancing reduz, mas não elimina, o impacto

Estratégias mais recentes de rebalance (cooperative sticky assignor) revogam apenas as partitions que realmente precisam mudar de dono, permitindo que consumers não afetados continuem processando durante a transição. Isso reduz a janela de impacto, mas não a elimina: as partitions que efetivamente trocam de consumer ainda passam por uma pausa de reatribuição.

Escalabilidade com Kubernetes

Escalar um Consumer Group horizontalmente (por exemplo, via HPA no Kubernetes, reagindo a consumer lag) tem uma consequência direta: cada evento de scale up ou scale down do deployment dispara um rebalance, porque o número de consumers do grupo muda. Um HPA configurado de forma agressiva (escalando e desescalando com frequência) pode gerar rebalances constantes, degradando o throughput que a escala deveria melhorar — o oposto do efeito pretendido.

Dica de entrevista

Se perguntarem sobre escalar consumers no Kubernetes, mencione que o número de partitions do tópico é o teto real de paralelismo — escalar o deployment além desse número não aumenta throughput, só produz consumers ociosos — e que HPAs devem ter cooldown adequado para não causar rebalances excessivos.

Quando isso vira problema arquitetural

- Dimensionar o número de partitions de um tópico sem considerar o número máximo de consumers que o time pretende escalar limita a escala futura sem uma migração de tópico (Capítulo 4).
- Processamento por mensagem mais lento que `max.poll.interval.ms` gera rebalances recorrentes, mesmo sem qualquer falha real de

infraestrutura — o sintoma parece instabilidade do Kafka, mas a causa é configuração de timeout desalinhada com o tempo de processamento real.
- Múltiplos Consumer Groups lendo o mesmo tópico não competem por partitions entre si — um rebalance em um grupo não afeta os demais grupos lendo o mesmo tópico.

Como aparece em entrevistas

"O que acontece quando um consumer cai?" é quase garantida em qualquer entrevista sobre Kafka em produção. A resposta fraca é "o Kafka redistribui as partitions automaticamente". A resposta forte explica o mecanismo: detecção via heartbeat/ `session.timeout.ms`, rebalance redistribuindo partitions entre os consumers restantes, e o impacto de pausa (parcial ou total, dependendo da estratégia de assignor) durante a transição.

Relação com Java e Spring Boot

Em Spring Kafka, o `group.id` é configurado no `@KafkaListener` (ou na configuração global do `ConsumerFactory`), e `concurrency` define quantas threads consumidoras a própria aplicação instancia internamente para esse listener — cada thread se comporta como um membro adicional do Consumer Group do ponto de vista do Kafka. Configurar `concurrency` maior que o número de partitions disponíveis para essa instância cria threads ociosas, pelo mesmo motivo do Cenário "Consumers > Partitions" acima.

> **Escalando o serviço de antifraude**
>
> O `antifraude-service` roda com 4 réplicas no Kubernetes, consumindo um tópico com 8 partitions — paralelismo saudável, 2 partitions por réplica. Em um Black Friday, o time aumenta para 12 réplicas esperando mais throughput, mas o tópico continua com 8 partitions: 4 réplicas ficam ociosas, e o throughput real não muda, porque o teto de paralelismo é o número de partitions, não o número de pods.

Resumo

Um Consumer Group garante que cada partition é processada por exatamente um consumer do grupo por vez, com o número de partitions definindo o teto de paralelismo. Rebalance redistribui partitions sempre que a composição do grupo muda — consumer entra, sai, ou é declarado morto por timeout — e tem um custo real de pausa, reduzido mas não eliminado por estratégias cooperativas. Escalar consumers além do número de partitions gera consumers ociosos, não mais throughput.

Pode vir a seguir

Prováveis follow-ups: "o que é `max.poll.interval.ms` e por que ele causa rebalances?", "qual a diferença entre eager e cooperative rebalancing?" e "como você monitoraria rebalances excessivos em produção?" (Capítulo 15).

PARTE III — CONSUMO E REPROCESSAMENTO

Offset e Commit

Os capítulos anteriores mencionaram offset como "a posição de um registro no log". Este capítulo entra no detalhe que mais aparece em incidentes de produção: como e quando essa posição é salva — e o que acontece quando ela é salva no momento errado.

Offset como posição de leitura

> **Offset**
>
> Offset é um número inteiro crescente, único por partition, que identifica a posição de um registro no log. Para um consumer, o offset também representa "até onde eu já li" — mas essa posição só é oficialmente registrada quando o consumer faz o commit dela.

É essencial separar dois conceitos que parecem a mesma coisa: a posição de leitura em memória do consumer (onde o client já buscou e entregou mensagens à aplicação) e o offset commitado (o que fica persistido no Kafka como "o Consumer Group processou até aqui"). Entre um e outro existe uma janela onde mensagens já foram entregues à aplicação, mas ainda não estão oficialmente marcadas como processadas.

offsets 4–6: já processados, ainda não commitados — reprocessados se o consumer cair agora

Offsets 0-3 já commitados; offsets 4-6 já processados pela aplicação, mas ainda não commitados; offset 7 ainda não lido.

Offset é por partition e por Consumer Group

> **Duas dimensões que não podem ser confundidas**
>
> O offset commitado não é uma propriedade global do tópico nem do consumer isoladamente — ele é armazenado por combinação de **partition** e **Consumer Group**. Isso significa que dois Consumer Groups diferentes lendo o mesmo tópico têm, cada um, seu próprio conjunto de offsets commitados, completamente independentes entre si.

É por isso que um novo Consumer Group, ao começar a consumir um tópico existente, não tem nenhum offset commitado ainda — e a configuração `auto.offset.reset` (`earliest` ou `latest`) decide se ele começa do início da retenção disponível ou apenas dos eventos novos a partir de agora.

Auto Commit

> **Auto Commit**
>
> Com `enable.auto.commit=true` (o padrão), o client Kafka comita automaticamente os offsets processados em um intervalo periódico (`auto.commit.interval.ms`), sem que a aplicação precise chamar nada explicitamente.

A vantagem é simplicidade: o desenvolvedor não escreve nenhum código relacionado a commit. O risco é que o commit acontece em um intervalo de tempo, não em sincronia com o processamento real da mensagem — é perfeitamente possível que o client comite um offset cujo processamento pela aplicação ainda está em andamento, ou pior, que falhou silenciosamente.

Commit Manual

> **Commit Manual**
>
> Com `enable.auto.commit=false`, a aplicação decide explicitamente quando comitar — tipicamente logo depois de confirmar que o processamento de uma mensagem (ou lote) terminou com sucesso, via `Acknowledgment.acknowledge()` no Spring Kafka ou `consumer.commitSync()` / `commitAsync()` no client puro.

Commit manual dá controle preciso sobre a relação entre "processei" e "commitei", ao custo de mais código e mais responsabilidade do desenvolvedor em acertar esse acoplamento.

Os riscos de cada abordagem

Auto commit vs. commit manual: o que pode dar errado

Ordem	Risco
Commit antes do processamento terminar	Se o consumer cair no meio do processamento, a mensagem é considerada processada mesmo sem ter sido — **perda de mensagem**
Processamento termina antes do commit	Se o consumer cair antes do commit acontecer, a mensagem será entregue de novo no próximo início — **duplicidade**

> **"Auto commit é mais seguro porque é automático"**
>
> Automático não significa seguro — significa que o timing do commit está fora do controle da aplicação. O auto commit tende ao cenário de **perda de mensagem**: ele

> pode comitar um offset cujo processamento ainda não terminou (ou falhou), porque o intervalo de commit não sabe nada sobre o estado real do processamento.

O commit manual, feito **depois** que o processamento termina com sucesso, elimina o risco de perda de mensagem, mas não elimina duplicidade: se o consumer cair exatamente entre "terminei de processar" e "commitei", a mensagem será reprocessada. Essa é a razão pela qual **at-least-once** (Capítulo 10) exige processamento idempotente (Capítulo 11) — a duplicidade não é um bug a ser eliminado, é uma possibilidade estrutural que o consumer precisa saber absorver.

Quando isso vira incidente real

Times que usam auto commit em processamento assíncrono ou multi-thread — onde a mensagem é entregue à aplicação e o processamento real acontece em outra thread ou fila interna — costumam descobrir tarde demais que o client já comitou o offset antes do processamento de fato terminar. Um crash da aplicação nesse intervalo perde a mensagem silenciosamente, sem exceção, sem log de erro — porque, do ponto de vista do Kafka, ela já foi processada.

> **Dica de entrevista**
>
> Ao explicar a diferença entre auto commit e commit manual, não pare em "um é automático e o outro manual" — explique a consequência: auto commit tende a perder mensagens sob falha, commit manual bem posicionado (depois do processamento) tende a duplicar mensagens sob falha. Nenhum dos dois é "mais seguro" em abstrato; o design correto depende de aceitar duplicidade e tratar processamento com idempotência.

Relação com Java e Spring Boot

Em Spring Kafka, `enable.auto.commit=false` combinado com `AckMode.MANUAL` (ou `MANUAL_IMMEDIATE`) no `ContainerFactory` é o padrão recomendado para processamento com garantias reais: o método anotado com `@KafkaListener` recebe um parâmetro `Acknowledgment`, e chama `acknowledgment.acknowledge()` apenas depois que toda a lógica de negócio da mensagem — incluindo qualquer escrita em banco — terminou com sucesso.

> **Processamento de boletos vencidos**
>
> O `cobranca-service` consome o tópico `boletos.vencidos` com commit manual: ele só chama `acknowledgment.acknowledge()` depois de confirmar que a notificação de cobrança foi enviada e registrada no banco. Se o serviço cair no meio do envio, o offset não avança — o boleto será reprocessado na próxima inicialização, gerando no máximo uma notificação duplicada (mitigada por uma checagem de idempotência, Capítulo 11), nunca uma cobrança perdida silenciosamente.

> **Resumo**
>
> Offset marca a posição de leitura em uma partition; commit é o que efetivamente persiste essa posição para um Consumer Group. Auto commit é simples, mas comita em um intervalo de tempo desalinhado do processamento real, tendendo a perder mensagens sob falha. Commit manual, feito após o processamento confirmado, tende a duplicar mensagens sob falha em vez de perdê-las — por isso processamento idempotente é um requisito, não um extra, em qualquer consumer com garantias at-least-once.

> **Pode vir a seguir**
>
> Prováveis follow-ups: "o que é `auto.offset.reset` e quando ele entra em ação?", "como você garantiria zero perda de mensagens no seu consumer?" e "o que é

> exactly-once e ele elimina a necessidade de idempotência?" (Capítulos 10 e 11).

PARTE III — CONSUMO E REPROCESSAMENTO

Retention e Replay

O Capítulo 1 usou retenção para explicar por que Kafka não é "só uma fila". Este capítulo aprofunda como essa retenção é configurada na prática e como ela vira a ferramenta mais poderosa (e mais arriscada) de correção de bugs em um sistema orientado a eventos: o replay.

Retention Period

> **Retention Period**
>
> Retention period é a política, configurada por tópico, que determina por quanto tempo (ou até que tamanho acumulado) os registros de uma partition permanecem no log antes de serem elegíveis para descarte.

Duas configurações controlam isso, e a que for atingida primeiro dispara a limpeza:

Retenção por tempo vs. por tamanho

Configuração	O que controla
`retention.ms`	Por quanto tempo um registro permanece disponível (ex.: 7 dias)
`retention.bytes`	Tamanho máximo acumulado por partition antes de descartar os registros mais antigos

> **Retenção não depende de consumo**

> Um registro não é removido por ter sido lido — ele é removido apenas quando expira pela política configurada, mesmo que nenhum consumer o tenha lido ainda. E o inverso também vale: um registro lido por todos os Consumer Groups existentes continua no log até a política de retenção expirá-lo, disponível para qualquer consumer futuro.

Replay: reprocessando o que já foi lido

> **Replay**
>
> Replay é o ato de mover a posição de leitura de um Consumer Group (ou de uma nova instância dele) para um ponto anterior no log — reprocessando eventos que já haviam sido consumidos, ou lendo pela primeira vez eventos antigos que um Consumer Group novo nunca viu.

Antes: offset commitado em 6. Depois de resetar o offset para 2, o Consumer Group reprocessa os offsets 2 a 6 novamente.

Tecnicamente, replay é simples: resetar o offset commitado de um Consumer Group para uma posição anterior (ou para o início da retenção disponível, com `--to-earliest`) e deixar o consumer avançar normalmente a partir dali. A complexidade não está em como fazer — está em saber quando fazer e quais efeitos colaterais esperar.

Para que replay serve

- **Corrigir um bug de processamento**: se uma versão com bug do `extrato-service` gerou lançamentos errados nos últimos 3 dias, corrigir o bug e resetar o offset para 3 dias atrás faz o serviço reconstruir esses lançamentos corretamente — assumindo que a lógica seja idempotente (Capítulo 11).
- **Popular um sistema novo com histórico existente**: um novo Consumer Group, ao assinar um tópico com `auto.offset.reset=earliest`, lê desde o início da retenção disponível — útil para inicializar um serviço de analytics ou um índice de busca sem precisar que o producer reenvie nada.
- **Reconstruir um estado derivado**: se um índice do Elasticsearch é populado a partir de eventos Kafka e precisa ser reconstruído do zero (mudança de schema, corrupção de índice), resetar o offset do Consumer Group responsável e deixá-lo reprocessar tudo reconstrói o índice sem tocar no restante do sistema.

Riscos do replay

> **"Replay é só resetar o offset, sem mais consequências"**
>
> Resetar o offset é a parte fácil. O replay reprocessa eventos que já geraram efeitos colaterais na primeira vez — notificações enviadas, e-mails disparados, saldos creditados. Se o consumer não for idempotente, o replay duplica esses efeitos: o cliente recebe a notificação de novo, o saldo é creditado uma segunda vez.

> **Replay em produção exige planejamento, não só o comando**
>
> Antes de fazer replay em um consumer de produção, é preciso confirmar que ele é idempotente (ou aceitar explicitamente os efeitos colaterais da duplicação), avisar sistemas downstream que dependem da ordem recente de processamento, e considerar o volume: reprocessar dias de eventos pode gerar um pico de carga equivalente a um dia inteiro de tráfego normal, concentrado em minutos.

Outra limitação prática: replay só alcança o que ainda não expirou. Um bug descoberto depois que a retenção já expirou os eventos afetados não pode ser corrigido por replay — a única saída, nesse caso, é reconstruir o estado a partir de outra fonte (backup, banco de dados, snapshot), se existir.

Como aparece em entrevistas

"Como o Kafka consegue reler mensagens já consumidas?" é uma variação comum, geralmente seguida de "e se eu precisasse corrigir um bug que afetou os últimos 2 dias de processamento, o que você faria?". A resposta esperada conecta retenção (só é possível reler o que ainda está retido), reset de offset (o mecanismo) e idempotência (a pré-condição para fazer isso com segurança em produção).

> **Dica de entrevista**
>
> Não descreva replay apenas como "resetar o offset" — mencione explicitamente a dependência da retenção (só funciona dentro da janela retida) e o requisito de idempotência do consumer, que é o que separa um replay seguro de um replay que duplica efeitos colaterais em produção.

Relação com Java e Spring Boot

Do lado da aplicação, replay geralmente não exige código novo — é uma operação administrativa feita via CLI (`kafka-consumer-groups.sh --reset-offsets`) ou via ferramentas de gestão de cluster, com o Consumer Group parado durante a operação. O código Spring Kafka do consumer não muda; o que muda é de onde ele recomeça a ler na próxima vez que subir. Por isso, a responsabilidade de tornar o replay seguro recai inteiramente sobre o design de idempotência do consumer, não sobre nenhuma configuração especial do client.

Reconstruindo o dashboard de risco

Um bug no cálculo de score de crédito do `risco-service` gerou scores incorretos nos últimos 5 dias. Depois de corrigir o bug, o time reseta o offset do Consumer Group `risco-service` para 5 dias atrás. Como o serviço grava o score mais recente por cliente (upsert, não incremento), reprocessar os mesmos eventos várias vezes é seguro — o replay simplesmente sobrescreve os scores errados pelos corretos, sem duplicar nada.

Resumo

Retention define por quanto tempo (ou até que tamanho) um evento permanece no log, independentemente de ter sido consumido. Replay usa essa retenção para reprocessar eventos — corrigindo bugs, populando sistemas novos ou reconstruindo estado derivado — movendo o offset commitado de um Consumer Group para um ponto anterior. É poderoso, mas só funciona dentro da janela de retenção disponível, e só é seguro em produção se o consumer for idempotente.

Pode vir a seguir

Prováveis follow-ups: "qual a diferença entre replay do Kafka e DLQ redrive do SQS?" e "como você faria replay em um consumer que não é idempotente, sem duplicar efeitos colaterais?".

PARTE III — CONSUMO E REPROCESSAMENTO

Retry e DLQ

O Capítulo 7 mostrou que commit manual, feito após o processamento, ainda deixa uma pergunta em aberto: o que fazer quando o processamento falha de verdade? Este capítulo fecha a Parte III respondendo isso — e encerrando um mal-entendido comum sobre DLQ no Kafka.

Erro transitório vs. erro permanente

> **Erro transitório**
>
> Um erro transitório é uma falha que tem chance real de não se repetir em uma nova tentativa — timeout de rede, banco de dados momentaneamente indisponível, serviço externo fora do ar por instantes.

> **Erro permanente**
>
> Um erro permanente é uma falha que vai se repetir independentemente de quantas vezes a mensagem for reprocessada — um payload malformado, um campo obrigatório ausente, uma regra de negócio que rejeita aquele dado especificamente.

Essa distinção é o que decide a estratégia: retry resolve erro transitório; DLQ existe para lidar com o que o retry não resolve.

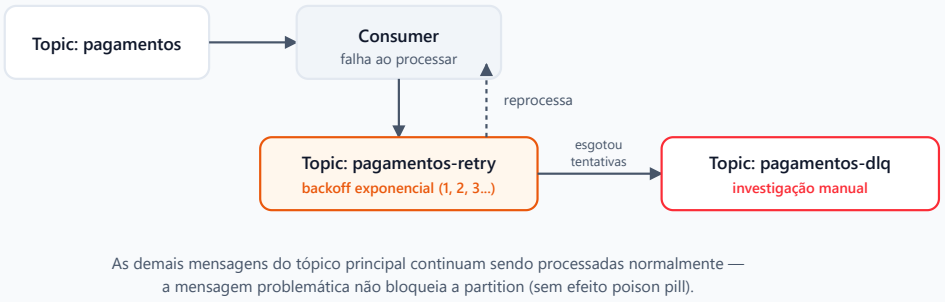
Retry e Backoff

> **Retry com Backoff**
>
> Retry é a nova tentativa de processar uma mensagem que falhou. Backoff é o intervalo, crescente a cada tentativa (backoff exponencial), entre uma tentativa e a

> próxima — evitando martelar um sistema já sobrecarregado com tentativas imediatas e sucessivas.

Um esquema comum: 3 tentativas, com intervalos de 1s, 5s e 30s. Se as três falharem, a mensagem é considerada não recuperável por retry simples e segue para a DLQ.

Retry Topic: como o Kafka implementa retry na prática



Consumer falha ao processar; a mensagem vai para um tópico de retry com backoff; após esgotar as tentativas, ela vai para a DLQ, sem bloquear as demais mensagens do tópico principal.

Diferente de um simples "tente de novo dentro do mesmo poll", o padrão mais robusto em Spring Kafka usa um **retry topic** dedicado: ao falhar, a mensagem é republicada em um tópico separado (`pagamentos-retry`), com um cabeçalho indicando quantas tentativas já ocorreram e quando a próxima deve acontecer. Um listener nesse tópico de retry reprocessa a mensagem após o backoff, e, se esgotar as tentativas configuradas, republica-a em um tópico de DLQ.

> **Por que não simplesmente bloquear e tentar de novo no mesmo poll**

Bloquear a thread do consumer tentando reprocessar a mesma mensagem repetidamente no tópico principal trava o processamento de todas as mensagens seguintes daquela partition — esse é exatamente o efeito poison pill. Usar um tópico de retry separado permite que o consumer principal siga processando as mensagens seguintes enquanto a mensagem problemática aguarda sua próxima tentativa em outro lugar.

Poison Pill

Poison Pill

Poison pill é uma mensagem que trava o processamento de uma partition indefinidamente — porque toda tentativa de processá-la falha e o consumer não tem estratégia para segui-la em frente, ficando preso reprocessando a mesma mensagem repetidamente.

Sem uma estratégia de retry com limite e DLQ, um consumer mal configurado que trata todo erro como transitório (retentando para sempre) pode ficar travado indefinidamente em uma única mensagem malformada, enquanto todas as mensagens seguintes da mesma partition esperam atrás dela na fila de processamento.

DLQ no Kafka: não é nativa

"O Kafka tem DLQ nativa" é uma afirmação incorreta

Diferente do AWS SQS, que tem um mecanismo de DLQ integrado à própria fila, o Kafka **não** tem um recurso nativo de dead-lettering. Uma DLQ no Kafka é, na prática, apenas um tópico comum, e a lógica de publicar mensagens problemáticas nele é responsabilidade da aplicação (ou de um framework como o Spring Kafka, que oferece o `DeadLetterPublishingRecoverer` pronto para uso).

Essa diferença é frequentemente mal respondida em entrevista — candidatos assumem que "Kafka tem DLQ" da mesma forma que SQS, quando na realidade é uma convenção arquitetural implementada por cima do Kafka, não um recurso da plataforma.

Replay vs. DLQ Redrive

Reprocessamento: replay (Kafka) vs. DLQ redrive (SQS)

Aspecto	Replay (Kafka)	DLQ Redrive (SQS)
O que reprocessa	Qualquer intervalo de eventos retidos no tópico original	Apenas as mensagens que já falharam e foram para a DLQ
Mecanismo	Reset do offset commitado do Consumer Group	Comando explícito de redrive movendo mensagens de volta à fila original
Escopo	Histórico completo dentro da retenção, processado ou não	Apenas o subconjunto que já falhou anteriormente

Replay e DLQ redrive resolvem problemas parecidos mas não idênticos: replay reconstrói processamento a partir de qualquer ponto do histórico retido; redrive apenas reencaminha mensagens que já haviam sido isoladas por falha.

Como aparece em entrevistas

"O Kafka tem DLQ nativa?" é uma pergunta direta e comum, quase sempre seguida de "como você implementaria uma?". A resposta correta explica que é um tópico comum com lógica de aplicação por trás, não um recurso de plataforma — e citar o `DeadLetterPublishingRecoverer` do Spring Kafka demonstra conhecimento prático, não só teórico.

Dica de entrevista

Ao ser perguntado sobre DLQ no Kafka, não responda apenas "sim, existe" — explique que é implementada como um tópico comum, por convenção da aplicação ou do framework, diferente do mecanismo nativo do SQS.

Relação com Java e Spring Boot

O Spring Kafka oferece `DefaultErrorHandler` combinado com `DeadLetterPublishingRecoverer` para implementar retry com backoff e DLQ de forma declarativa: configura-se o número de tentativas, o backoff entre elas, e o recoverer publica automaticamente no tópico de DLQ (por convenção, `<topico-original>.DLT`) quando as tentativas se esgotam — sem que o desenvolvedor precise escrever a lógica de republicação manualmente.

Boleto com CPF inválido

Um evento `BoletoGerado` com CPF malformado falha na validação do `cobranca-service` — um erro permanente, não transitório. Mesmo com retry configurado, as 3 tentativas falham da mesma forma, e a mensagem é publicada em `boletos.gerados.DLT` para investigação manual, enquanto os boletos seguintes da mesma partition continuam sendo processados normalmente.

Resumo

Erros transitórios justificam retry com backoff; erros permanentes não são resolvidos por retry e devem ir para uma DLQ. O Kafka não tem DLQ nativa — é um tópico comum, populado por convenção da aplicação ou framework, existindo para isolar o efeito poison pill sem bloquear o processamento das demais mensagens. Replay (Kafka) e DLQ redrive (SQS) resolvem reprocessamento de formas diferentes: um reconstrói a partir de qualquer ponto do histórico retido, o outro reencaminha apenas o que já foi isolado por falha.

Pode vir a seguir

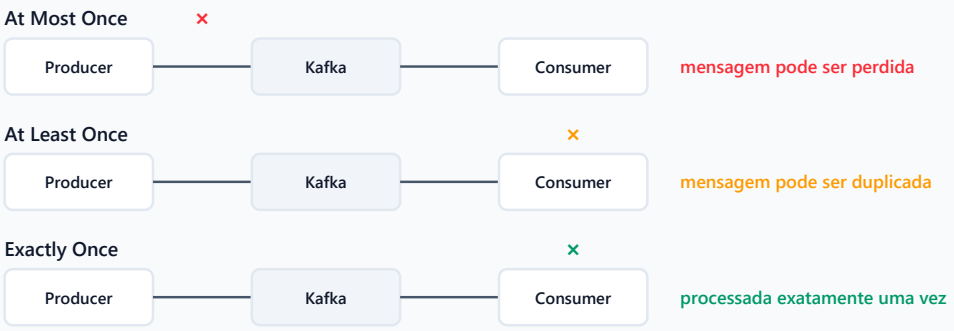
Prováveis follow-ups: "como você decidiria quantas tentativas de retry fazer antes de ir para a DLQ?" e "o que você faria com as mensagens que já estão na DLQ?".

PARTE IV — CONFIABILIDADE

Garantias de entrega

Os Capítulos 7 e 9 já tocaram em perda e duplicidade de mensagens, cada um do seu ângulo (commit, retry). Este capítulo nomeia formalmente os três contratos que resumem esse comportamento — e desfaz um mal-entendido comum sobre o que "exactly-once" realmente cobre.

As três garantias



At Most Once perde mensagens sob falha antes da confirmação; At Least Once duplica mensagens sob falha após o processamento; Exactly Once, com idempotência e transações, garante processamento único mesmo sob falha.

At Most Once

A mensagem é entregue no máximo uma vez — nunca duplicada, mas pode ser perdida. Acontece quando o commit (ou o ack do producer) ocorre antes de qualquer confirmação de sucesso do passo seguinte.

At Least Once

A mensagem é entregue pelo menos uma vez — nunca perdida, mas pode ser duplicada. Acontece quando o commit (ou retry do producer) só ocorre depois que

o passo anterior foi confirmado, aceitando reprocessamento em caso de falha entre a confirmação e o registro dela.

Exactly Once

A mensagem é processada exatamente uma vez do ponto de vista do efeito observável — nem perdida, nem duplicada — através de uma combinação de producer idempotente, transações e consumer idempotente, não de uma garantia mágica de entrega única.

Onde a garantia é decidida

Do lado do producer, a configuração `acks` decide o trade-off entre disponibilidade e durabilidade (Capítulo 5): `acks=0` favorece velocidade e aceita perda; `acks=all` favorece durabilidade. Do lado do consumer, a ordem entre processamento e commit (Capítulo 7) decide entre perda (commit antes de processar) e duplicidade (commit depois de processar, mas com risco de falha no meio).

Configuração típica por garantia

Garantia	Producer	Consumer
At Most Once	`acks=0` ou `acks=1` sem retry	Auto commit, ou commit antes do processamento
At Least Once	`acks=all` com retry do producer	Commit manual, depois que o processamento termina
Exactly Once	Producer idempotente + transações	Consumer idempotente, ou transação que abrange leitura+escrita

Por que At Least Once é o padrão prático

> **At Most Once raramente é aceitável em sistemas financeiros**
>
> Perder uma mensagem silenciosamente — um evento de pagamento que nunca é processado — é geralmente pior do que processá-lo duas vezes, porque a duplicidade pelo menos é detectável e corrigível com idempotência, enquanto a perda sequer deixa rastro. Por isso, a configuração mais comum em sistemas financeiros é At Least Once, combinada com processamento idempotente no consumer.

Isso explica por que a idempotência (Capítulo 11) não é um "extra de qualidade" — ela é o complemento necessário do At Least Once, que é a garantia mais usada na prática.

Exactly Once: o que ela cobre e o que não cobre

> **"Exactly Once significa que a mensagem nunca é processada duas vezes, ponto final"**
>
> Exactly-once semantics (EOS) do Kafka garante processamento exatamente uma vez **dentro do ecossistema Kafka** — entre um tópico de entrada, o processamento, e a escrita em outro tópico Kafka, usando transações. No momento em que o efeito sai do Kafka — uma chamada HTTP, uma escrita em banco de dados externo, um e-mail enviado — essa garantia deixa de valer automaticamente, porque esses sistemas externos não participam da transação do Kafka.

> **Exactly-once não elimina a necessidade de idempotência no mundo real**
>
> Se o consumer escreve em um banco de dados ou chama uma API externa como parte do processamento, essa escrita fica fora do escopo da transação do Kafka. Uma falha exatamente depois de processar mas antes de Kafka confirmar o commit

da transação ainda pode levar a uma repetição desse efeito externo. Por isso, sistemas que integram Kafka com bancos de dados e APIs externas continuam precisando de idempotência nessas integrações, mesmo usando exactly-once semantics internamente.

Como aparece em entrevistas

"O que é exactly-once e ele resolve duplicidade de vez?" é a armadilha mais comum desta seção. A resposta fraca diz "sim, elimina duplicidade". A resposta forte explica o escopo: EOS cobre o fluxo Kafka-para-Kafka via transações, mas qualquer efeito colateral fora do Kafka (banco, API, e-mail) continua exigindo idempotência própria.

Dica de entrevista

Ao ser perguntado sobre exactly-once, sempre delimite o escopo: "dentro do Kafka, entre ler de um tópico e escrever em outro, via transações". Isso sinaliza que você entende a fronteira da garantia, não só o nome.

Relação com Java e Spring Boot

Em Spring Kafka, `acks=all` e `enable.idempotence=true` no producer (o padrão desde versões recentes do client) evitam duplicação de mensagens causada por retries automáticos do próprio producer. Para exactly-once semantics completo entre tópicos, usa-se `KafkaTransactionManager` envolvendo leitura e escrita em uma mesma transação Kafka. Mas assim que o listener chama um serviço HTTP externo ou grava em um banco relacional fora dessa transação, a responsabilidade de evitar duplicidade volta a ser do desenvolvedor — tipicamente via constraint de unicidade e checagem de idempotência (Capítulo 11).

Notificação de PIX recebido

O `notificacao-service` roda com At Least Once: se ele cair depois de enviar a notificação push mas antes de comitar o offset, o evento será reprocessado e uma segunda notificação pode ser enviada. Isso é aceitável para uma notificação (o pior caso é o cliente receber dois avisos), mas seria inaceitável para o `saldo-service`, que precisa de proteção explícita contra crédito duplicado — daí a diferença de rigor de idempotência entre os dois consumers do mesmo evento.

Resumo

At Most Once nunca duplica, mas pode perder. At Least Once nunca perde, mas pode duplicar — e é o padrão prático em sistemas financeiros, combinado com idempotência. Exactly Once, via transações do Kafka, garante processamento único apenas dentro do ecossistema Kafka; qualquer efeito colateral externo (banco, API, e-mail) continua exigindo idempotência própria, independentemente da garantia configurada internamente.

Pode vir a seguir

Prováveis follow-ups: "como você evitaria processamento duplicado?" (Capítulo 11) e "qual configuração de `acks` você usaria para um tópico de transações financeiras, e por quê?".

PARTE IV — CONFIABILIDADE

Idempotência

O Capítulo 10 terminou apontando para cá: At Least Once — a garantia mais usada em sistemas financeiros — aceita duplicidade como possibilidade estrutural. Este capítulo mostra como um consumer absorve essa duplicidade sem gerar efeitos colaterais duplicados.

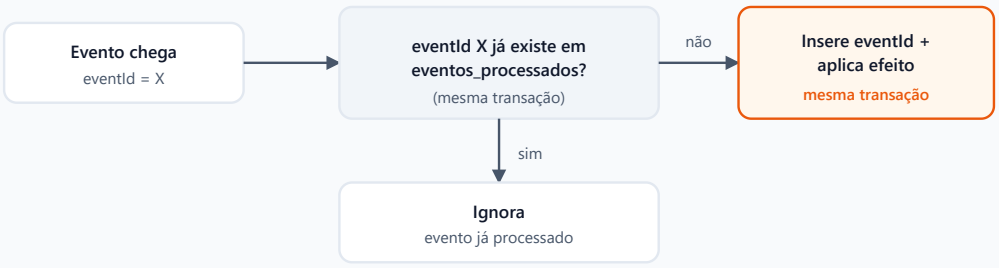
O que é idempotência

> **Idempotência**
>
> Um processamento é idempotente quando produz o mesmo efeito final, independentemente de ser executado uma ou várias vezes com a mesma entrada. Para um consumer Kafka, isso significa: processar o mesmo evento duas vezes não deve gerar o dobro do efeito — o saldo é creditado uma vez, o e-mail é enviado uma vez, o registro existe uma vez.

Por que consumers precisam ser idempotentes

> **Duplicidade não é uma possibilidade remota**
>
> At Least Once (Capítulo 10) garante que nenhuma mensagem é perdida, aceitando como preço que ela pode ser entregue mais de uma vez — depois de um rebalance (Capítulo 6), de um retry de commit, ou de um crash entre o fim do processamento e o commit do offset (Capítulo 7). Isso não é um bug raro: é o comportamento esperado e documentado dessa garantia. Um consumer que não trata isso vai duplicar efeitos em produção, mais cedo ou mais tarde.

O padrão: eventId + constraint de unicidade



Constraint de unicidade em eventId garante atomicidade mesmo sob concorrência.

O consumer verifica se o eventId já existe na tabela de eventos processados, na mesma transação que aplica o efeito de negócio.

O padrão mais comum e mais robusto: cada evento carrega um identificador único (`eventId` , geralmente um UUID gerado pelo producer no momento da criação do evento). O consumer mantém uma tabela (`eventos_processados` , ou similar) com uma constraint de unicidade nesse `eventId` . Antes de aplicar o efeito de negócio, o consumer tenta inserir o `eventId` nessa tabela; se a inserção falhar por violação de unicidade, o evento já foi processado e a operação é ignorada com segurança.

Atomicidade: a parte que costuma ser esquecida

> **"Eu checo se o eventId já existe antes de processar"**
>
> Checar e processar em passos separados (uma consulta `SELECT` , depois um `INSERT / UPDATE` de negócio) introduz uma condição de corrida: sob concorrência (dois consumers, ou reprocessamento simultâneo), ambos podem checar "não existe" ao mesmo tempo e aplicar o efeito duas vezes antes que qualquer um grave o `eventId` . A checagem e a aplicação do efeito precisam acontecer na **mesma transação de banco**, com a constraint de unicidade como a garantia real — não uma checagem prévia em código de aplicação.

Na prática, isso significa: inserir o `eventId` na tabela de controle e aplicar o efeito de negócio (debitar, creditar, gravar) na mesma transação. Se a constraint de unicidade rejeitar a inserção, a transação inteira falha e é revertida — nenhum efeito parcial é aplicado, mesmo sob concorrência.

eventId vs. transactionId

Duas chaves de deduplicação diferentes

Chave	Identifica	Uso típico
`eventId`	O evento Kafka específico (uma publicação)	Deduplicar reentrega do mesmo evento (retry, rebalance)
`transactionId` (ou ID de negócio)	A transação de domínio (ex.: o PIX, o pagamento)	Deduplicar a operação de negócio, mesmo se vier de eventos diferentes

Normalmente `eventId` é suficiente para lidar com reentrega do Kafka. Mas em cenários onde o mesmo fato de negócio pode chegar por caminhos diferentes (ex.: uma notificação do banco duplicada pelo canal de origem, gerando dois eventos Kafka distintos para a mesma transação real), a deduplicação por `eventId` sozinha não basta — é necessário também um identificador de negócio único (`transactionId` , `endToEndId` no caso de PIX) para pegar essa duplicidade de origem, antes mesmo de chegar ao Kafka.

Quando idempotência natural dispensa a tabela de controle

Nem todo consumer precisa da tabela `eventos_processados`. Se a operação já é naturalmente idempotente — um *upsert* que sempre grava o mesmo valor final, independentemente de quantas vezes for executado — o controle explícito de `eventId` é redundante.

> **Upsert de score de crédito**
>
> O `risco-service` grava o score mais recente de cada cliente com um `UPDATE ... WHERE clienteId = ?` (upsert). Processar o mesmo evento duas vezes simplesmente grava o mesmo valor duas vezes — sem efeito colateral adicional. Esse consumer é naturalmente idempotente e não precisa de controle de `eventId`.

Já operações que **incrementam** ou **acumulam** estado (creditar saldo, incrementar contador, enviar notificação) não são naturalmente idempotentes — processá-las duas vezes duplica o efeito, e por isso exigem o controle explícito de `eventId`.

Como aparece em entrevistas

"Como você evitaria processamento duplicado?" é praticamente garantida sempre que At Least Once entra na conversa. A resposta fraca menciona apenas "eventId único". A resposta forte explica a atomicidade: checagem e aplicação do efeito na mesma transação, com constraint de unicidade como garantia real sob concorrência — não uma checagem prévia em código.

> **Dica de entrevista**
>
> Ao descrever idempotência, sempre mencione a constraint de unicidade e a atomicidade da transação. É esse detalhe — não a existência de um `eventId` —

> que separa uma implementação correta de uma que ainda tem condição de corrida sob concorrência.

Relação com Java e Spring Boot

Em Spring Boot, isso tipicamente se traduz em uma tabela com uma coluna `event_id` marcada `UNIQUE`, e o código de processamento dentro de um método anotado com `@Transactional`, que tenta inserir o registro de controle e aplicar o efeito de negócio antes de qualquer commit. Se a inserção lançar uma `DataIntegrityViolationException` (violação da constraint), o código captura essa exceção especificamente e trata como "já processado" — não como um erro genérico a ser propagado para o mecanismo de retry (Capítulo 9).

> **Crédito de PIX idempotente**
>
> ```
> @Transactional
> public void creditarPix(PixRecebidoEvent evento) {
>     try {
>         eventosProcessadosRepository.insert(evento.getEventId(
>     } catch (DataIntegrityViolationException e) {
>         return; // já processado, nada a fazer
>     }
>     contaRepository.creditar(evento.getContaId(), evento.getVa
> }
> ```
>
> A inserção do `eventId` e o crédito do saldo acontecem na mesma transação — se a inserção falhar por duplicidade, o crédito nunca é aplicado uma segunda vez.

> **Resumo**
>
> Idempotência é o que permite um consumer absorver a duplicidade estrutural do At Least Once sem duplicar efeitos de negócio. O padrão central é um `eventId` único,

verificado e inserido na mesma transação que aplica o efeito — a constraint de unicidade do banco, não uma checagem prévia em código, é o que garante atomicidade sob concorrência. Operações naturalmente idempotentes (upsert) podem dispensar esse controle; operações que acumulam estado (creditar, incrementar) não podem.

Pode vir a seguir

Prováveis follow-ups: "o que você faria se o mesmo evento de negócio chegasse por dois `eventId` diferentes?" e "como a transactional outbox se relaciona com idempotência?" (Capítulo 12).

PARTE IV — CONFIABILIDADE

Transações e Outbox Pattern

A pergunta 10 já levantou o problema: o que acontece se a escrita no banco tiver sucesso mas a publicação no Kafka falhar (ou vice-versa)? Este capítulo fecha a Parte IV respondendo isso — e mostra por que a resposta não é "usar uma transação distribuída".

O problema do Dual Write

> **Dual Write**
>
> Dual write é o padrão (problemático) de escrever em dois sistemas diferentes — um banco de dados e o Kafka — como duas operações separadas e não atômicas entre si. Se uma tiver sucesso e a outra falhar, os dois sistemas ficam inconsistentes.

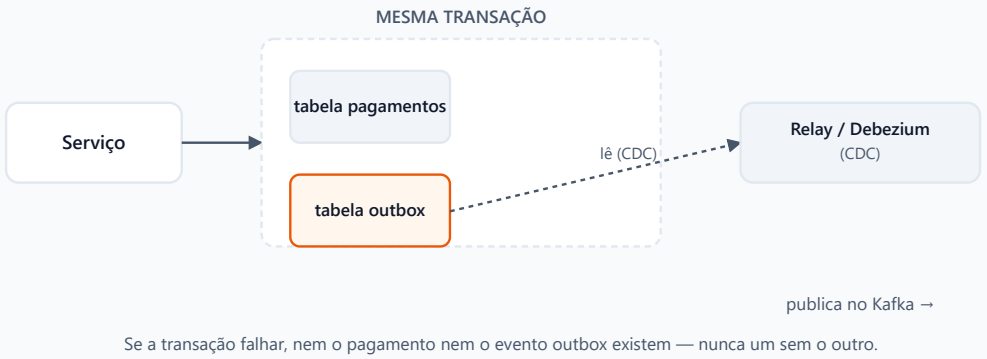
Um `pagamento-service` típico, ao aprovar um pagamento, faz duas coisas: grava o resultado no banco (tabela `pagamentos`) e publica um evento `PagamentoAprovado` no Kafka. Se essas duas escritas não forem atômicas, existem duas janelas de falha:

- O banco é gravado, mas a publicação no Kafka falha (rede, broker indisponível) — o pagamento existe, mas nenhum sistema downstream (notificação, contabilidade, antifraude) fica sabendo.
- O evento é publicado no Kafka, mas a gravação no banco falha depois — sistemas downstream reagem a um pagamento que, na verdade, nunca foi persistido com sucesso.

> **Kafka e bancos relacionais não compartilham uma transação nativa**
>
> Não existe uma transação distribuída nativa e simples entre um banco de dados relacional e o Kafka (XA/2PC entre os dois é tecnicamente possível em teoria, mas praticamente não utilizado em produção pela complexidade operacional e impacto

de performance). O dual write, feito de forma ingênua, é uma fonte garantida de inconsistência em produção.

Transactional Outbox: a solução



O serviço grava a tabela de negócio e a tabela outbox na mesma transação de banco. Um processo relay (CDC) lê a tabela outbox e publica os eventos no Kafka.

Transactional Outbox

Transactional outbox é o padrão que resolve o dual write escrevendo a intenção de publicar um evento em uma tabela auxiliar (`outbox`) **na mesma transação de banco** que grava o efeito de negócio. Um processo separado — o relay — lê essa tabela e publica os eventos no Kafka de forma assíncrona e confiável.

A ideia central: como a tabela de negócio e a tabela outbox são gravadas na mesma transação relacional (algo que o banco já garante nativamente), elas nunca ficam inconsistentes entre si — ou as duas existem, ou nenhuma existe. O problema de "publicar no Kafka de forma confiável" vira "ler uma tabela e publicar", que é mais simples de tornar robusto (com retry, at-least-once) do que coordenar duas escritas heterogêneas.

Como o relay lê a tabela outbox

Duas abordagens comuns:

Estratégias de relay para a tabela outbox

Estratégia	Como funciona
Polling	Um processo consulta periodicamente a tabela outbox por registros não publicados, publica no Kafka, e marca como publicado
CDC (Change Data Capture)	Uma ferramenta como o Debezium lê o log de transações do próprio banco (WAL/binlog) e publica automaticamente qualquer inserção na tabela outbox, sem polling

Dica de entrevista

CDC via Debezium é considerado a abordagem mais robusta porque lê diretamente o log de transações do banco — não depende de uma consulta periódica com atraso, e captura o evento no exato momento em que a transação é confirmada. Mencionar isso demonstra conhecimento além do padrão em si.

Outbox garante publicação — não elimina duplicidade

"Outbox pattern resolve tudo, inclusive duplicidade"

Outbox garante que o evento **será** publicado se a transação de negócio foi confirmada — resolve o problema de publicação perdida (o cenário "banco gravado, Kafka não publicado"). Ele não impede que o mesmo evento seja publicado mais de uma vez (por exemplo, se o relay falhar depois de publicar mas antes de marcar como publicado, e tentar de novo). O consumer do outro lado ainda precisa ser

idempotente (Capítulo 11) — outbox e idempotência resolvem problemas complementares, não substituem um ao outro.

Quando usar

- Sempre que uma operação de negócio precisa, de forma confiável, tanto persistir estado quanto notificar outros sistemas via evento — o caso mais comum em arquiteturas orientadas a eventos com banco relacional.
- Especialmente importante quando a ausência do evento (silenciosamente) seria pior do que um evento duplicado — que é o caso da maioria dos sistemas financeiros.

Quando não vale o custo

Para operações onde a publicação do evento não é crítica (perder ocasionalmente é aceitável, ex.: um evento de analytics de baixo valor), o custo operacional de manter uma tabela outbox e um relay pode não se justificar frente a uma publicação direta simples, aceitando a rara janela de inconsistência.

Relação com Java e Spring Boot

Em Spring Boot, a implementação mais comum grava a tabela outbox usando o mesmo `EntityManager`/transação JPA da operação de negócio, dentro do mesmo método `@Transactional`. O relay geralmente é um processo separado — seja um scheduler simples (`@Scheduled`) fazendo polling, seja um conector Debezium configurado para monitorar a tabela outbox e publicar automaticamente no Kafka, sem código adicional na aplicação.

Aprovação de pagamento com outbox

O `pagamento-service` grava, na mesma transação, a linha em `pagamentos` (status = aprovado) e uma linha em `outbox` com o payload do evento

`PagamentoAprovado`. Se a transação falhar por qualquer motivo, nenhuma das duas é persistida — não existe cenário onde o pagamento é aprovado sem o evento correspondente ser eventualmente publicado, nem o inverso. O Debezium, monitorando a tabela outbox via CDC, publica o evento no Kafka assim que a transação é confirmada.

Resumo

Dual write — escrever em banco e Kafka como operações separadas — é uma fonte estrutural de inconsistência, porque as duas escritas não compartilham uma transação nativa. Transactional outbox resolve isso gravando a intenção do evento na mesma transação da operação de negócio, delegando a publicação real a um processo relay (polling ou CDC via Debezium). Outbox garante que o evento será publicado, mas não elimina a possibilidade de duplicidade — o consumer ainda precisa ser idempotente.

Pode vir a seguir

Prováveis follow-ups: "por que não usar uma transação distribuída (2PC) entre o banco e o Kafka?" e "como você monitoraria se o relay da tabela outbox está atrasado ou travado?" (Capítulo 15).

Producer com Spring Kafka

Os capítulos anteriores explicaram conceitos que valem para qualquer client Kafka. A partir daqui, a Parte V mostra como esses conceitos aparecem em código Java real, usando Spring Kafka — começando pelo lado que publica eventos.

KafkaTemplate

> **KafkaTemplate**
>
> `KafkaTemplate<K, V>` é a abstração do Spring Kafka para publicar mensagens — o equivalente, para produção de eventos, do `JdbcTemplate` para acesso a banco. Ele encapsula o `KafkaProducer` nativo do client Java, expondo uma API mais simples e integrada ao restante do Spring (injeção de dependência, propriedades de configuração, métricas).

```
@Service
public class PagamentoEventPublisher {

    private final KafkaTemplate<String, PagamentoAprovadoEvent> ka

    public PagamentoEventPublisher(KafkaTemplate<String, Pagamento
        this.kafkaTemplate = kafkaTemplate;
    }

    public void publicar(PagamentoAprovadoEvent evento) {
        kafkaTemplate.send("pagamentos.aprovados", evento.getConta
    }
}
```

Topic, Key e Value

O `send` mais comum recebe três argumentos: o tópico, a key (Capítulo 4 — decide a partition de destino e a ordenação relativa) e o value (o payload do evento). Overloads com dois argumentos (`send(topic, value)`) publicam sem key, sujeitos à distribuição aproximadamente uniforme entre partitions.

> **Nunca esqueça a key em eventos que exigem ordenação**
>
> Um refactor que remove a key "para simplificar o código" silenciosamente destrói a ordenação relativa entre eventos da mesma entidade (Capítulo 4). Trate a key como parte do contrato do tópico, documentada junto com ele — não como um detalhe implícito do código do producer.

Serialização

O Spring Kafka delega a serialização de key e value a classes `Serializer<T>` configuradas via `spring.kafka.producer.key-serializer` / `value-serializer` . As opções mais comuns em produção:

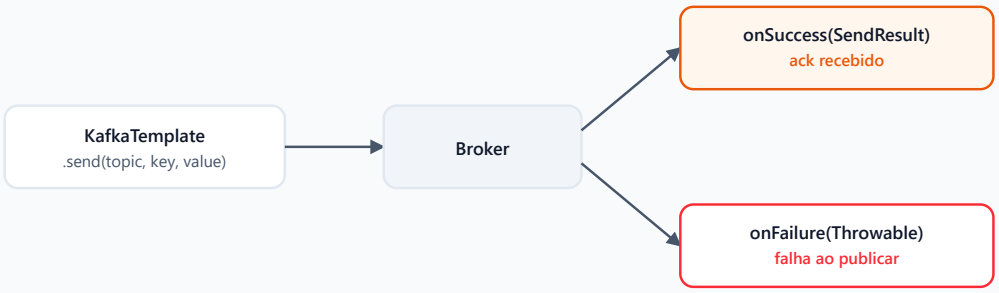
Estratégias de serialização

Formato	Característica
JSON (`JsonSerializer`)	Simples, legível, mas sem verificação de schema — mudanças incompatíveis só quebram em runtime
Avro (com Schema Registry)	Schema versionado e compatibilidade verificada em build/publish time; payload binário compacto
Protobuf	Similar ao Avro em garantias de schema, com forte tipagem no Java gerado a partir do `.proto`

Times que ainda não têm Schema Registry geralmente começam com JSON e migram para Avro/Protobuf quando o número de consumidores cresce o suficiente para que mudanças incompatíveis de payload se tornarem um risco real de quebra entre serviços de times diferentes.

Callbacks: tratando o resultado do envio

`send()` é assíncrono — retorna um `CompletableFuture<SendResult<K, V>>` imediatamente, sem esperar a confirmação do broker. Ignorar esse retorno significa não saber se a publicação teve sucesso.



KafkaTemplate.send() é assíncrono; o resultado chega via callback, onSuccess com o SendResult ou onFailure com a exceção.

```
kafkaTemplate.send("pagamentos.aprovados", evento.getContaId(), ev
    .whenComplete((result, exception) -> {
        if (exception != null) {
            log.error("Falha ao publicar evento de pagamento", exc
        } else {
            log.debug("Evento publicado no offset {}", result.getR
        }
    });
```

> **"Chamei o send(), então o evento foi publicado"**
>
> Chamar `send()` apenas enfileira a mensagem para envio assíncrono — não garante que ela chegou ao broker. Sem tratar o callback (ou usar `.get()` de forma

síncrona, com o custo de latência que isso implica), falhas de publicação passam silenciosamente despercebidas, exatamente o cenário de dual write discutido no Capítulo 12.

Headers e correlationId

Além de key e value, uma mensagem Kafka carrega **headers** — metadados arbitrários, tipicamente usados para rastreabilidade e integração entre serviços, sem poluir o payload de negócio.

```
ProducerRecord<String, PagamentoAprovadoEvent> record =
    new ProducerRecord<>("pagamentos.aprovados", evento.getContaId
record.headers().add("correlationId", correlationId.getBytes(Stand

kafkaTemplate.send(record);
```

Dica de entrevista

Mencionar `correlationId` propagado via headers demonstra conhecimento prático de observabilidade em sistemas distribuídos: ele permite rastrear uma requisição desde a chamada HTTP original, passando pelo evento Kafka, até os consumers que o processam — essencial para debugar um fluxo que atravessa múltiplos serviços (aprofundado no Capítulo 15).

Tratamento de falhas do producer

Falhas de publicação se dividem nas mesmas categorias do Capítulo 9: transitórias (broker temporariamente indisponível — o client Kafka já faz retry automático internamente, configurável via `retries` e `delivery.timeout.ms`) e permanentes (mensagem maior que `max.request.size`, serialização que lança exceção). O callback `onFailure` é o lugar certo para logar, alertar, ou — em casos críticos — persistir a mensagem falha para reenvio manual, já que o `send()` não bloqueia a aplicação esperando esse resultado.

Como aparece em entrevistas

"Como você garantiria que um evento foi publicado com sucesso no Kafka?" é a pergunta prática mais comum sobre producer. A resposta esperada cita o retorno assíncrono do `send()`, o tratamento do callback (não apenas chamar e seguir em frente), e — para o cenário de consistência com banco de dados — o outbox pattern do Capítulo 12, já que nenhum callback de producer resolve o problema de dual write sozinho.

Relação com sistemas reais

> **Publicando compra autorizada com rastreabilidade**
>
> O `autorizacao-service`, ao aprovar uma compra no cartão, publica o evento `CompraAutorizada` usando `cartaoId` como key (ordenação por cartão, Capítulo 4), propaga o `correlationId` da requisição HTTP original via header, e trata o callback do `send()` registrando uma métrica de falha de publicação — que alimenta um alerta se a taxa de falha ultrapassar um limiar (Capítulo 15).

> **Resumo**

> `KafkaTemplate` encapsula o producer Kafka no Spring Boot: `send(topic, key, value)` publica de forma assíncrona, retornando um `CompletableFuture` que deve ser tratado via callback para saber se a publicação teve sucesso. Serialização (JSON, Avro, Protobuf) e headers (como `correlationId`, para rastreabilidade) completam o contrato de uma publicação bem-feita. Nenhum desses mecanismos, isoladamente, resolve consistência com o banco de dados — para isso, o padrão é o outbox (Capítulo 12).

> **Pode vir a seguir**
>
> Prováveis follow-ups: "o que acontece se o `send()` falhar silenciosamente sem callback tratado?" e "como você propagaria um `correlationId` de uma chamada HTTP até o consumer do outro lado do Kafka?".

Consumer com Spring Kafka

O Capítulo 13 mostrou o lado que publica. Este capítulo mostra o lado que consome — onde a maioria dos conceitos das Partes II a IV (Consumer Group, offset/commit, retry/DLQ, idempotência) se materializa em código real.

@KafkaListener

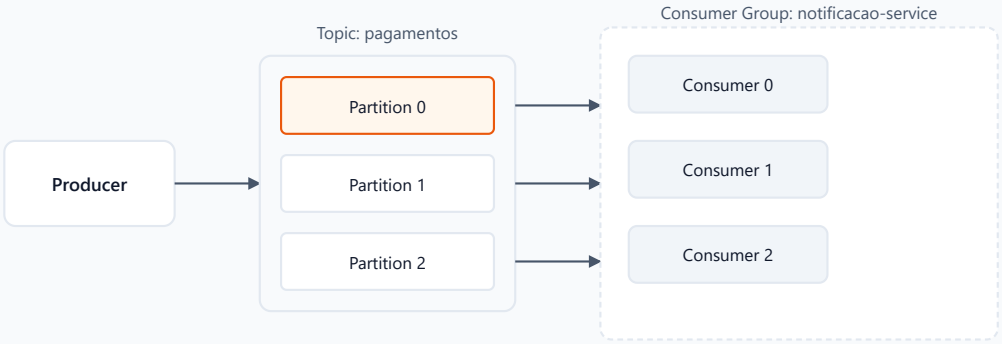
> **@KafkaListener**
>
> `@KafkaListener` é a anotação do Spring Kafka que registra um método como consumer de um ou mais tópicos. Por trás dela, o Spring Kafka gerencia o ciclo de vida completo do `KafkaConsumer` nativo — poll loop, deserialização, e (dependendo da configuração) commit de offset.

```
@Component
public class PagamentoEventListener {

    @KafkaListener(topics = "pagamentos.aprovados", groupId = "not
    public void processar(PagamentoAprovadoEvent evento) {
        notificacaoService.enviarPush(evento.getContaId(), evento.
    }
}
```

`groupId` define o Consumer Group (Capítulo 6) — se omitido no método, cai no valor padrão de `spring.kafka.consumer.group-id` da aplicação.

Concurrency



Cada partition do tópico é atribuída a exatamente um consumer do Consumer Group — concurrency define quantas threads a própria instância da aplicação registra como consumers adicionais.

```
@KafkaListener(
    topics = "pagamentos.aprovados",
    groupId = "notificacao-service",
    concurrency = "3"
)
public void processar(PagamentoAprovadoEvent evento) { /* ... */ }
```

Concurrency não ultrapassa o número de partitions

`concurrency = "3"` cria 3 threads consumidoras dentro da mesma instância da aplicação, cada uma se comportando como um membro adicional do Consumer Group. Configurar `concurrency` maior que o número de partitions disponíveis para essa instância cria threads ociosas — o mesmo limite do Capítulo 6 se aplica aqui, só que dentro de um único processo em vez de entre múltiplos pods.

Acknowledgment: commit manual na prática

> **Acknowledgment**
>
> `Acknowledgment` é o objeto que o Spring Kafka injeta no método do listener quando o `AckMode` está configurado como manual, permitindo à aplicação decidir explicitamente quando o offset deve ser commitado.

```
@KafkaListener(topics = "pagamentos.aprovados", groupId = "notifica
public void processar(PagamentoAprovadoEvent evento, Acknowledgmen
    notificacaoService.enviarPush(evento.getContaId(), evento.getV
    ack.acknowledge();
}
```

Isso exige, na configuração do `ContainerFactory`, `enable.auto.commit=false` e `AckMode.MANUAL` (ou `MANUAL_IMMEDIATE`) — o padrão discutido no Capítulo 7, aqui aplicado em código: `ack.acknowledge()` só é chamado depois que a lógica de negócio (aqui, o envio da notificação) termina com sucesso.

Desserialização

O `value-deserializer` configurado (JSON, Avro, Protobuf — espelhando a escolha do producer no Capítulo 13) converte o payload bruto de volta para o tipo Java esperado pelo método do listener. Uma incompatibilidade de schema entre o que o producer serializou e o que o consumer espera desserializar é uma fonte comum de erro **permanente** (Capítulo 9) — a mensagem nunca vai desserializar corretamente, não importa quantas vezes seja tentada.

Tratamento de exceções: retry e DLQ na prática

```
@Bean
public DefaultErrorHandler errorHandler(KafkaTemplate<Object, Obje
    var recoverer = new DeadLetterPublishingRecoverer(template);
    var backoff = new ExponentialBackOff(1000L, 2.0);
    backoff.setMaxInterval(30_000L);
    return new DefaultErrorHandler(recoverer, backoff);
}
```

Esse `DefaultErrorHandler`, registrado no `ContainerFactory`, implementa exatamente o padrão do Capítulo 9: tentativas com backoff exponencial e, ao esgotá-las, publicação automática no tópico de DLQ (por convenção, `<topico>.DLT`) via `DeadLetterPublishingRecoverer` — sem que o método do listener precise tratar isso manualmente.

> **"Uma exceção não tratada no listener derruba a aplicação"**
>
> Uma exceção lançada dentro do método `@KafkaListener` não derruba a aplicação — ela é capturada pelo container do Spring Kafka e encaminhada ao `ErrorHandler` configurado (retry, depois DLQ). O erro real de não configurar um `ErrorHandler` adequado não é a aplicação cair, é o comportamento padrão (retry indefinido, ou log genérico) não ser o que o time espera para aquele fluxo de negócio.

Idempotência no consumer

Todo o cuidado de commit manual e retry não substitui a idempotência (Capítulo 11) — eles resolvem *quando* o offset avança e *o que fazer* quando o processamento falha, não *o que acontece* se a mesma mensagem for entregue duas vezes.

```
@KafkaListener(topics = "pix.recebido", groupId = "saldo-service")
@Transactional
public void processar(PixRecebidoEvent evento, Acknowledgment ack)
    try {
        eventosProcessadosRepository.insert(evento.getEventId());
    } catch (DataIntegrityViolationException e) {
        ack.acknowledge();
        return;
    }
    contaRepository.creditar(evento.getContaId(), evento.getValor(
    ack.acknowledge();
}
```

Como aparece em entrevistas

"Como você implementaria um consumer Kafka em Spring Boot com garantias de at-least-once e proteção contra duplicidade?" é uma pergunta que amarra praticamente todo o livro. A resposta completa cita: `@KafkaListener` com `groupId` explícito, `AckMode.MANUAL` com `Acknowledgment.acknowledge()` após o processamento, `DefaultErrorHandler` com backoff e DLQ, e checagem de `eventId` na mesma transação do efeito de negócio.

> **Dica de entrevista**
>
> Se o entrevistador pedir para "desenhar" um consumer completo, monte a resposta nessa ordem: listener e group, ack manual, error handler com retry/DLQ, idempotência. Essa sequência espelha exatamente a jornada dos Capítulos 6 a 11 — e mostra que os conceitos se conectam, não são tópicos isolados.

Relação com sistemas reais

> **Consumer completo do saldo-service**

O `saldo-service` roda com `concurrency = "6"` (igual ao número de partitions do tópico `pix.recebido`), `AckMode.MANUAL`, um `DefaultErrorHandler` com backoff de 1s/5s/30s e DLQ, e checagem de `eventId` na mesma transação do crédito. O resultado: nenhuma mensagem perdida (at-least-once), nenhum crédito duplicado (idempotência), e mensagens problemáticas isoladas na DLQ sem travar o processamento das demais contas.

Resumo

`@KafkaListener` registra o consumer; `groupId` define o Consumer Group; `concurrency` cria threads adicionais dentro da instância, limitadas pelo número de partitions disponíveis. `Acknowledgment` viabiliza commit manual após o processamento. `DefaultErrorHandler` com `DeadLetterPublishingRecoverer` implementa retry com backoff e DLQ declarativamente. Nenhum desses mecanismos substitui a checagem de idempotência via `eventId` na mesma transação do efeito de negócio — eles são complementares, não alternativos.

Pode vir a seguir

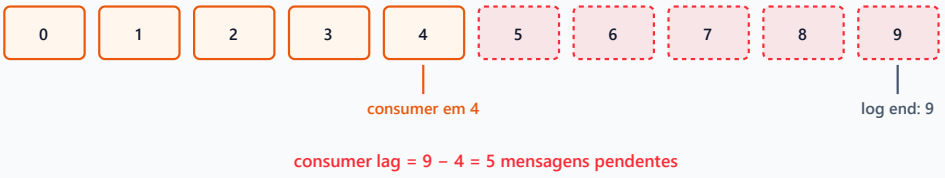
Prováveis follow-ups: "o que acontece se `ack.acknowledge()` for chamado antes da exceção ser lançada?" e "como você testaria esse consumer, incluindo o cenário de mensagem duplicada?".

Observabilidade

Este último capítulo fecha o livro amarrando um fio que atravessou quase todos os anteriores: como saber, em produção, se tudo que foi descrito nos capítulos anteriores está de fato funcionando — sem esperar um incidente para descobrir que não estava.

Consumer Lag: o sinal vital mais importante

> **Consumer Lag**
>
> Consumer lag é a diferença entre o offset mais recente já produzido em uma partition (log end offset) e o offset commitado pelo Consumer Group. Representa quantas mensagens já existem no tópico mas ainda não foram processadas.



O producer já escreveu até o offset 9; o consumer só processou até o offset 4. O consumer lag é a diferença: 5 mensagens pendentes.

Lag zero (ou próximo de zero, oscilando com o tráfego normal) indica um consumer saudável. Lag que cresce de forma sustentada ao longo do tempo é o sinal mais confiável de que algo está errado — processamento lento, rebalances excessivos (Capítulo 6), ou um erro fazendo o consumer ficar preso reprocessando (Capítulo 9).

Lag momentâneo não é o mesmo que lag crescente

Um pico de lag durante um aumento pontual de tráfego é esperado e geralmente se resolve sozinho. O sinal de alerta real é a **tendência**: lag que não volta a cair depois que o pico de tráfego passa, ou que cresce de forma constante ao longo de horas — isso indica que o throughput de consumo é estruturalmente menor que o de produção.

Throughput e taxa de erro

Throughput

Throughput é o volume de mensagens processadas (ou produzidas) por unidade de tempo. Combinado com consumer lag, é o par de métricas que diagnostica a saúde de capacidade de um pipeline: throughput de consumo menor que throughput de produção, sustentado, é a causa direta de lag crescente.

Taxa de erro (proporção de mensagens que falham no processamento, incluindo as que acabam na DLQ do Capítulo 9) complementa esse par: um consumer pode ter throughput alto mas taxa de erro também alta, processando rápido mas incorretamente — daí a importância de monitorar as duas métricas juntas, não isoladas.

Logs estruturados e correlationId

Log genérico vs. log estruturado com correlationId

Abordagem	Exemplo	Problema/vantagem
Log genérico	`"Erro ao processar evento"`	Impossível rastrear qual evento, qual conta, qual fluxo — inútil sob volume
Log estruturado	`{"correlationId": "abc-123", "eventId": "...", "contaId": "...", "erro": "timeout"}`	Permite filtrar e correlacionar logs de múltiplos serviços pelo mesmo `correlationId`

O `correlationId`, introduzido no Capítulo 13 como header do evento, é o que permite reconstruir o caminho completo de uma requisição — da chamada HTTP original, passando pela publicação no Kafka, até cada consumer que processou o evento — em ferramentas de agregação de log como Kibana ou Datadog.

Métricas essenciais para um pipeline Kafka

Métricas que valem a pena alertar

Métrica	Por quê
Consumer lag por partition/grupo	Detecta degradação de throughput antes que vire incidente visível ao usuário
Taxa de mensagens na DLQ	Sinaliza falhas persistentes que exigem investigação manual
Taxa de rebalance	Rebalances frequentes indicam `max.poll.interval.ms` mal configurado ou instabilidade de infraestrutura (Capítulo 6)
Latência producer → consumer	Tempo entre publicação e processamento, relevante para SLAs de negócio
Taxa de erro por consumer	Complementa throughput — processar rápido e errado não é melhor que processar devagar

Tracing distribuído

Além de logs e métricas, tracing distribuído (OpenTelemetry, integrado ao Spring Boot via Micrometer Tracing) permite visualizar o caminho de uma transação como um grafo de spans — a chamada HTTP, a publicação Kafka, o processamento em cada consumer — com duração de cada etapa. Isso é particularmente valioso para diagnosticar *onde*, num fluxo que atravessa múltiplos serviços via Kafka, a maior parte da latência está sendo gasta.

Ferramentas do ecossistema

Onde cada ferramenta entra

Ferramenta	Papel típico
Prometheus	Coleta e armazena métricas (consumer lag, throughput, taxa de erro) expostas via Micrometer
Grafana	Visualização de dashboards e configuração de alertas sobre as métricas do Prometheus
Kibana	Busca e visualização de logs estruturados (tipicamente com stack ELK/OpenSearch)
Datadog / New Relic	Plataformas comerciais que unificam métricas, logs e tracing em um único produto

Nenhuma dessas ferramentas é exigida pelo Kafka em si — a JVM do Spring Boot expõe métricas via Micrometer, e a escolha de backend (Prometheus+Grafana open source, ou uma plataforma comercial) é uma decisão de infraestrutura, não do design do pipeline de eventos.

Como aparece em entrevistas

"Como você monitoraria Kafka em produção?" é quase sempre a última pergunta de uma entrevista sobre o tema, testando se o candidato pensa em operação, não só em código. A resposta forte não lista ferramentas soltas — conecta métrica a decisão: "eu alertaria sobre consumer lag crescente, porque isso indica throughput insuficiente antes que o usuário sinta o atraso; e sobre taxa de mensagens na DLQ, porque isso indica falhas persistentes que passaram do retry".

> **Dica de entrevista**

Ao responder sobre observabilidade, sempre conecte a métrica ao sintoma de negócio que ela antecipa. "Eu monitoro consumer lag" é uma resposta incompleta; "eu monitoro consumer lag porque ele antecipa atraso de processamento antes que o usuário perceba" demonstra que você pensa em operação real, não em uma lista de métricas decorada.

Relação com sistemas reais

Alertando sobre atraso no crédito de PIX

O time de pagamentos configura um alerta no Grafana: se o consumer lag do `saldo-service` no tópico `pix.recebido` ultrapassar 1000 mensagens por mais de 2 minutos, um alerta é disparado no canal de plantão — porque, para esse fluxo específico, mais que alguns segundos de atraso entre o recebimento do PIX e o crédito do saldo já é um problema visível para o cliente.

Resumo

Observabilidade de um pipeline Kafka combina consumer lag (o atraso de processamento), throughput e taxa de erro (capacidade e qualidade), logs estruturados com correlationId (rastreabilidade entre serviços) e tracing distribuído (visão fim-a-fim de latência). Nenhuma métrica isolada conta a história completa — é a combinação delas, conectada a sintomas de negócio reais, que permite detectar problemas antes que o usuário os sinta.

Pode vir a seguir

Prováveis follow-ups: "conte um projeto real utilizando Kafka" e "quais erros ou desafios você já enfrentou com Kafka" — as duas últimas perguntas do livro, que pedem para conectar tudo isso a uma experiência concreta, não a teoria.