# **Apache Kafka: Uma Aula Completa sobre a Plataforma de Streaming de Eventos**

## **1. Introdução ao Apache Kafka**

O Apache Kafka é uma plataforma de streaming de eventos distribuída, de código aberto, desenvolvida para lidar com grandes volumes de dados em tempo real.1 Originado no LinkedIn em 2011 por Jay Kreps, Neha Narkhede e Jun Rao, o Kafka foi criado para suprir a necessidade de um sistema com alta taxa de transferência (throughput) e baixa latência, capaz de gerenciar quantidades massivas de dados de eventos em tempo real.1 Posteriormente, foi doado à Apache Software Foundation, onde se tornou um projeto de alto nível.2 O nome "Kafka" foi uma escolha de Jay Kreps, inspirado no autor Franz Kafka, por ser "um sistema otimizado para escrita" e por sua apreciação pela obra do autor.2

O problema fundamental que o Kafka buscou resolver foi a complexidade e a ineficiência na movimentação de grandes volumes de dados entre diferentes sistemas e aplicações. Antes do Kafka, as empresas frequentemente recorriam a uma variedade de sistemas de mensageria ponto a ponto ou soluções customizadas, o que resultava em uma arquitetura de integração frágil, difícil de escalar e manter. O Kafka propôs uma abordagem unificada, atuando como um "sistema nervoso central" para os dados de uma organização.

Inicialmente concebido como um sistema de enfileiramento de mensagens otimizado, o Kafka evoluiu significativamente para se tornar uma plataforma completa de streaming de eventos.1 Essa evolução foi impulsionada pela crescente demanda por processamento de dados em tempo real e pela necessidade de aplicações que pudessem reagir instantaneamente a novos eventos. Hoje, mais de 80% das empresas da Fortune 500, incluindo gigantes como Target, Microsoft, AirBnB e Netflix, utilizam o Kafka para fornecer experiências em tempo real orientadas por dados.1

Um diferencial crucial do Kafka em relação aos sistemas de mensageria tradicionais é a **retenção de mensagens**. Enquanto as filas de mensagens convencionais geralmente excluem as mensagens após serem consumidas, o Kafka armazena as mensagens em um log durável e de acréscimo (append-only log) por um período configurável.1 Isso permite que múltiplas aplicações consumidoras leiam os mesmos dados independentemente e em momentos diferentes, e até mesmo reprocessem dados históricos.3 Essa capacidade de retenção é fundamental para casos de uso como

*event sourcing*, processamento de fluxos e construção de pipelines de dados resilientes, transformando o Kafka no padrão de fato para streaming de eventos em tempo real.1

## **2. Conceitos Fundamentais do Kafka**

Para compreender o funcionamento do Apache Kafka, é essencial conhecer seus componentes e conceitos arquiteturais principais.

### **2.1. Arquitetura: Brokers, Clusters, Tópicos, Partições e Offsets**

A arquitetura do Kafka é projetada para escalabilidade, tolerância a falhas e alto desempenho. Seus principais elementos são:

* **Brokers:** Um broker Kafka é, essencialmente, um servidor Kafka.4 Cada broker é responsável por receber mensagens de produtores, armazená-las e servi-las a consumidores. Os brokers gerenciam o armazenamento e a distribuição de mensagens.4
* **Cluster:** Um cluster Kafka consiste em um ou mais brokers trabalhando em conjunto.6 A natureza distribuída do cluster permite alta disponibilidade e tolerância a falhas. Se um broker falhar, outros brokers no cluster podem assumir suas responsabilidades, garantindo a continuidade do serviço.3
* **Tópicos (Topics):** Um tópico é uma categoria ou nome de feed para o qual os registros (mensagens) são publicados.4 Os tópicos no Kafka são fluxos de registros particionados. Pense em um tópico como uma tabela em um banco de dados ou uma pasta em um sistema de arquivos; eles são usados para organizar as mensagens. Por exemplo, um tópico pode ser  
  pedidos\_clientes ou logs\_servidor\_web.8
* **Partições (Partitions):** Cada tópico no Kafka é dividido em uma ou mais partições.8 As partições são a unidade fundamental de paralelismo no Kafka. Uma partição é uma sequência ordenada e imutável de mensagens, onde cada mensagem na partição recebe um ID sequencial chamado offset.8 As mensagens são adicionadas ao final de uma partição (append-only). A divisão de um tópico em múltiplas partições permite que o Kafka distribua a carga de leitura e escrita entre vários brokers e consumidores, possibilitando o processamento paralelo.8
* **Offsets:** Um offset é um identificador numérico sequencial e imutável que o Kafka atribui a cada mensagem dentro de uma partição.8 O offset identifica unicamente uma mensagem dentro de sua partição. Os consumidores utilizam os offsets para rastrear sua posição de leitura em cada partição. Por exemplo, se um consumidor leu até o offset 25, o próximo offset a ser lido e, subsequentemente, "commitado" (confirmado) será o 26.10 Os offsets garantem a ordenação das mensagens  
  *dentro de uma partição*.2

**Relação entre Tópicos, Partições e Offsets:**

A estrutura de tópicos, partições e offsets é o que permite ao Kafka alcançar alta escalabilidade e garantir a ordem das mensagens (dentro de cada partição). Um tópico é um agrupamento lógico de dados. Para permitir o processamento paralelo e a distribuição de carga, esse tópico é dividido em partições. Cada partição funciona como um log independente e ordenado. Dentro de cada partição, as mensagens são identificadas e ordenadas por seus offsets.8

Essa arquitetura permite que múltiplos produtores escrevam em diferentes partições de um mesmo tópico simultaneamente, e múltiplos consumidores (geralmente organizados em grupos de consumidores) leiam de diferentes partições em paralelo. A ordenação das mensagens é garantida apenas no nível da partição; não há garantia de ordem global entre todas as partições de um tópico, a menos que o tópico tenha apenas uma partição (o que limita o paralelismo).2

Tipos de Offsets Relevantes:

Existem alguns termos específicos para offsets que são cruciais para o funcionamento e monitoramento do Kafka 8:

* **Log-End Offset (LEO):** Representa o offset da última mensagem presente em uma partição. É o final do log da partição.8
* **High Watermark (HW):** É o offset da última mensagem que foi replicada com sucesso para todas as réplicas *in-sync* (sincronizadas) de uma partição. Os consumidores só podem ler mensagens até o High Watermark, garantindo que leiam apenas dados que foram confirmados como duráveis.8
* **Committed Offset:** Indica o offset da última mensagem que um consumidor (ou grupo de consumidores) processou com sucesso e confirmou (commitou) de volta ao Kafka. Esses offsets são armazenados em um tópico interno do Kafka chamado \_\_consumer\_offsets.8 O committed offset é crucial para a tolerância a falhas, permitindo que um consumidor reinicie do ponto onde parou.8 É importante notar que o offset commitado refere-se ao  
  *próximo* offset que o consumidor pretende ler, e não ao offset da última mensagem processada com sucesso.10

O monitoramento de offsets, especialmente a diferença entre o Log-End Offset e o Committed Offset (conhecido como *consumer lag*), é vital para identificar problemas de desempenho no cluster Kafka.8

### **2.2. Produtores, Consumidores e Grupos de Consumidores (Consumer Groups)**

Os principais atores que interagem com o Kafka são os Produtores e os Consumidores:

* **Produtores (Producers):** São aplicações cliente responsáveis por publicar (escrever) fluxos de registros (mensagens) em um ou mais tópicos Kafka.4 O produtor não precisa se preocupar com qual consumidor irá ler a mensagem; ele apenas a envia para o tópico apropriado. Produtores podem escolher para qual partição enviar uma mensagem específica, seja explicitamente ou usando uma chave de partição que o Kafka usará para determinar a partição (geralmente por hashing da chave).12 Se nenhuma chave ou partição for especificada, as mensagens são distribuídas entre as partições de forma round-robin por padrão.12  
    
  Para se conectar a um cluster Kafka, os produtores precisam de configurações como bootstrap.servers (uma lista de endereços de brokers para a descoberta inicial do cluster) e serializadores para a chave e o valor da mensagem (ex: key.serializer, value.serializer).4
* **Consumidores (Consumers):** São aplicações cliente responsáveis por se inscrever (ler) em um ou mais tópicos Kafka e processar o fluxo de registros produzidos para esses tópicos.4 Um consumidor lê as mensagens de uma partição específica em ordem, usando o offset para rastrear sua posição.  
    
  Assim como os produtores, os consumidores requerem configurações como bootstrap.servers e desserializadores para a chave e o valor da mensagem (ex: key.deserializer, value.deserializer).4 Após a configuração, o consumidor se inscreve em um ou mais tópicos e entra em um loop infinito, continuamente requisitando (polling) novas mensagens.11 Os registros consumidos (ConsumerRecords) contêm a chave, valor, partição, timestamp e headers opcionais.11
* **Grupos de Consumidores (Consumer Groups):** Para escalar o consumo de mensagens e permitir o processamento paralelo, o Kafka utiliza o conceito de Grupos de Consumidores.11 Um grupo de consumidores é um conjunto de instâncias de consumidores que compartilham um  
  group.id comum e cooperam para consumir mensagens de um ou mais tópicos.11  
    
  O Kafka distribui as partições de um tópico entre os consumidores de um mesmo grupo, garantindo que cada partição seja consumida por exatamente um consumidor dentro do grupo.11 Isso significa que se um tópico tem 4 partições e um grupo de consumidores tem 2 consumidores, cada consumidor lerá de 2 partições. Se houver 4 consumidores, cada um lerá de uma partição. Se houver mais consumidores do que partições (ex: 5 consumidores para 4 partições), um consumidor ficará ocioso.14  
    
  Essa atribuição de partições permite o paralelismo no consumo. Se um consumidor em um grupo falhar ou se novos consumidores forem adicionados ao grupo, o Kafka automaticamente aciona um processo de rebalanceamento, redistribuindo as partições entre os membros ativos do grupo.11 Esse rebalanceamento garante a tolerância a falhas e a elasticidade no consumo.  
    
  É importante destacar que diferentes grupos de consumidores podem consumir do mesmo tópico de forma independente, cada um mantendo seus próprios offsets.11 Isso permite que múltiplos sistemas processem o mesmo fluxo de eventos para diferentes propósitos sem interferir uns nos outros, já que o consumo de dados não os apaga do Kafka (até que o período de retenção expire).11  
  O gerenciamento dos grupos de consumidores, incluindo a atribuição de partições e o rastreamento do consumo, é facilitado por um **Coordenador de Grupo (Group Coordinator)**, que é um broker Kafka eleito para essa função.14 Os consumidores enviam sinais periódicos (heartbeats) ao coordenador para indicar que estão ativos. Se um consumidor para de enviar heartbeats, o coordenador o considera falho e inicia um rebalanceamento.14

### **2.3. Gerenciamento de Metadados: ZooKeeper vs. KRaft**

O gerenciamento de metadados é crucial para a operação de um cluster Kafka. Os metadados incluem informações sobre brokers, tópicos, partições, configurações, ACLs (Listas de Controle de Acesso) e liderança de partições. Historicamente, o Kafka dependia de um serviço de consenso externo chamado Apache ZooKeeper para essa tarefa.15

* Modo ZooKeeper:  
  No modo tradicional, um broker no cluster Kafka é eleito como o Controlador (Controller). Este controlador é responsável por monitorar o estado dos outros brokers, gerenciar as réplicas de partições e comunicar-se com o ZooKeeper para ler e escrever metadados.16 O ZooKeeper atuava como o repositório persistente para os metadados do cluster.16  
    
  Embora funcional, a dependência do ZooKeeper adicionava complexidade operacional, pois exigia a manutenção de um sistema distribuído separado. Além disso, o ZooKeeper podia se tornar um gargalo em clusters muito grandes, limitando a escalabilidade em termos de número de partições e a velocidade de recuperação em caso de falhas do controlador.16
* Modo KRaft (Kafka Raft Metadata mode):  
  Com o lançamento do Apache Kafka 3.3.1 em outubro de 2022, um novo protocolo de consenso para gerenciamento de metadados, chamado KRaft, foi marcado como pronto para produção.16 O modo KRaft elimina a necessidade de executar um cluster ZooKeeper separado junto com cada cluster Kafka.15  
    
  No KRaft, um subconjunto de brokers Kafka, designados como Controladores de Quórum (Quorum Controllers), assume a responsabilidade pelo gerenciamento de metadados usando o protocolo de consenso Raft.15 Os metadados do cluster são agora armazenados em um tópico interno do Kafka chamado  
  \_\_cluster\_metadata.16 Os brokers que não são controladores continuam a lidar com as solicitações de dados, enquanto os controladores gerenciam as solicitações relacionadas a metadados.15  
    
  Um cluster KRaft pode operar em modo dedicado (nós são apenas brokers ou apenas controladores) ou compartilhado (alguns nós desempenham ambas as funções), dependendo do tamanho e das necessidades do cluster.16

Vantagens do KRaft sobre o ZooKeeper:

O modo KRaft oferece várias vantagens significativas 16:

1. **Implantação e Administração Mais Simples:** Ao consolidar o gerenciamento de metadados dentro do próprio Kafka, a pegada operacional é reduzida. Há apenas uma aplicação para instalar, configurar, proteger e gerenciar, simplificando a administração e facilitando a implantação do Kafka em ambientes com recursos limitados, como dispositivos de borda (edge).
2. **Escalabilidade Aprimorada:** O KRaft permite que o Kafka escale para um número muito maior de partições (milhões em um único cluster), em comparação com as dezenas de milhares que eram o limite prático com o ZooKeeper. O tempo de recuperação de falhas do controlador também é significativamente mais rápido (uma ordem de magnitude) com KRaft.
3. **Propagação de Metadados Mais Eficiente:** A propagação de metadados baseada em log e orientada a eventos no KRaft resulta em melhor desempenho para muitas das funções principais do Kafka, como a eleição de líderes de partição e rebalanceamentos de grupos de consumidores.

A migração de um cluster existente do modo ZooKeeper para o modo KRaft é suportada, e diversas configurações relacionadas ao ZooKeeper foram removidas ou alteradas no modo KRaft.15 Por exemplo, configurações de shutdown gracioso de brokers e versões de protocolo entre brokers são gerenciadas de forma diferente ou se tornaram obsoletas.15 Plugins como

CreateTopicPolicy e AlterConfigPolicy agora rodam nos controladores KRaft em vez dos brokers.15

## **3. Fluxo de Mensagem no Kafka: Passo a Passo**

Compreender o ciclo de vida de uma mensagem, desde sua criação pelo produtor até seu consumo e confirmação pelo consumidor, é fundamental para entender como o Kafka opera.

Passo 1: Configuração do Produtor

O processo inicia com a configuração da aplicação produtora. Propriedades essenciais incluem 4:

* bootstrap.servers: Uma lista de endereços de brokers (host:porta) que o produtor usará para a conexão inicial e descoberta do restante do cluster Kafka.
* key.serializer: Classe usada para serializar a chave da mensagem (ex: org.apache.kafka.common.serialization.StringSerializer para chaves string).
* value.serializer: Classe usada para serializar o valor (corpo) da mensagem (ex: org.apache.kafka.common.serialization.StringSerializer para valores string).

Passo 2: Publicação de Mensagens pelo Produtor

Uma vez configurado, o produtor cria um ProducerRecord, que encapsula o tópico de destino, opcionalmente uma chave, e o valor da mensagem.4

* ProducerRecord record = new ProducerRecord("meuTopico", "chaveOpcional", "Olá, Kafka!");  
  O produtor então envia este registro usando o método send().4
* producer.send(record);  
  O envio é geralmente assíncrono. O produtor pode especificar um callback para ser notificado sobre o sucesso ou falha da entrega.17  
    
  A mensagem é enviada para um broker líder da partição de destino. Se uma chave é fornecida, o Kafka usa um hash da chave para determinar a partição (garantindo que mensagens com a mesma chave vão para a mesma partição). Se nenhuma chave é fornecida, as mensagens são distribuídas de forma round-robin entre as partições do tópico (comportamento padrão) ou conforme uma estratégia de particionamento customizada.12

Passo 3: Processamento e Armazenamento pelo Broker

O broker líder da partição recebe a mensagem do produtor.

* Ele anexa a mensagem ao final do log da partição.
* Atribui um offset único à mensagem dentro daquela partição.
* Se a replicação estiver configurada (o que é o padrão para durabilidade), o líder replica a mensagem para os brokers seguidores (followers) da mesma partição.
* Uma vez que o número mínimo de réplicas sincronizadas (min.insync.replicas) tenha confirmado o recebimento da mensagem (e se o produtor estiver configurado com acks=all), o líder envia uma confirmação (acknowledgment) de volta ao produtor.18

Passo 4: Configuração do Consumidor

Similarmente ao produtor, a aplicação consumidora precisa ser configurada 4:

* bootstrap.servers: Lista de endereços de brokers para conexão inicial.
* key.deserializer: Classe para desserializar a chave da mensagem.
* value.deserializer: Classe para desserializar o valor da mensagem.
* group.id: Identificador único para o grupo de consumidores ao qual este consumidor pertence. Essencial para consumo paralelo e rebalanceamento.11
* auto.offset.reset: Define o que fazer quando não há um offset inicial no Kafka ou se o offset atual não existe mais no servidor (ex: earliest para começar do início, latest para começar do fim).

Passo 5: Consumo de Mensagens pelo Consumidor

O consumidor se inscreve (subscribe) em um ou mais tópicos.4

* consumer.subscribe(Arrays.asList("meuTopico"));  
  O consumidor então entra em um loop, chamando repetidamente o método poll() para buscar mensagens do broker.4
* ConsumerRecords records = consumer.poll(Duration.ofMillis(100));  
  O método poll() retorna um conjunto de ConsumerRecords para as partições atribuídas ao consumidor. Cada ConsumerRecord contém a mensagem (chave, valor), o tópico, a partição e o offset da mensagem.11  
    
  A aplicação consumidora itera sobre esses registros e processa cada mensagem.11

Passo 6: Commit de Offset pelo Consumidor

Após processar com sucesso um lote de mensagens, o consumidor precisa "commitar" os offsets. Commitar um offset significa informar ao Kafka que as mensagens até aquele offset (para uma dada partição e grupo de consumidores) foram processadas com sucesso.8

O Kafka armazena esses offsets commitados em um tópico interno especial chamado \_\_consumer\_offsets.10

Existem duas principais formas de commit de offset:

1. **Auto-commit:** O consumidor pode ser configurado para commitar offsets automaticamente em intervalos regulares (controlado por enable.auto.commit=true e auto.commit.interval.ms). Embora conveniente, pode levar à perda de mensagens (se o consumidor falhar após o commit, mas antes do processamento completo) ou ao processamento duplicado (se falhar antes do commit, mas após o processamento).10
2. **Manual commit:** A aplicação tem controle explícito sobre quando os offsets são commitados, usando consumer.commitSync() (síncrono, bloqueante) ou consumer.commitAsync() (assíncrono, não bloqueante).4 O commit manual oferece maior controle sobre as garantias de entrega (pelo menos uma vez, exatamente uma vez – este último geralmente requer gerenciamento de estado adicional).

Se um consumidor falhar e reiniciar, ou se um rebalanceamento ocorrer, o novo consumidor (ou o mesmo consumidor reiniciado) usará o último offset commitado para aquela partição para continuar o processamento de onde parou, evitando a perda de dados ou o reprocessamento desnecessário de grandes volumes de dados.8

Este fluxo garante que as mensagens sejam movidas de forma eficiente dos produtores para os consumidores, com o Kafka atuando como um intermediário durável e escalável.4

## **4. Capacidades Chave do Kafka**

O Apache Kafka se destaca por um conjunto de capacidades que o tornam ideal para arquiteturas de dados modernas e orientadas a eventos.

### **4.1. Alta Vazão (Throughput) e Baixa Latência**

Kafka é projetado para alcançar alta vazão (a taxa na qual o sistema pode processar mensagens) e baixa latência (o atraso entre a produção e o consumo de uma mensagem).1 Vários mecanismos contribuem para isso:

1. **I/O Sequencial e Logs de Partição:** O Kafka armazena mensagens em disco de forma sequencial em logs de partição (append-only logs). As escritas sequenciais em disco são significativamente mais rápidas do que as escritas aleatórias, pois minimizam o movimento da cabeça do disco. Da mesma forma, as leituras sequenciais são otimizadas pelo sistema operacional através de técnicas como *read-ahead*. Essa estrutura de log é fundamental para o desempenho do Kafka.2
2. **Técnica 'Zero-Copy':** Para transferir dados do arquivo de log da partição para a rede (para os consumidores), o Kafka utiliza a técnica 'zero-copy' sempre que possível. Em sistemas operacionais que suportam essa funcionalidade (como Linux com a chamada de sistema sendfile()), os dados são transferidos diretamente do cache de página do sistema operacional para o buffer da placa de rede, sem passar pelo espaço de usuário da aplicação Kafka (broker). Isso reduz drasticamente o número de cópias de dados na memória e as trocas de contexto entre o kernel e o espaço do usuário, diminuindo a sobrecarga da CPU e da memória, e aumentando a taxa de processamento de mensagens.20
3. **Agrupamento de Mensagens (Batching):**
   * **Produtores:** Os produtores Kafka podem agrupar múltiplas mensagens em lotes (batches) antes de enviá-las ao broker. Isso é controlado pelos parâmetros batch.size (tamanho do lote em bytes) e linger.ms (tempo máximo que o produtor espera para preencher um lote).19 Enviar mensagens em lotes maiores reduz a sobrecarga da rede (menos requisições) e melhora a compressão (se habilitada), resultando em maior vazão.19
   * **Consumidores:** Os consumidores também podem buscar mensagens em lotes dos brokers, controlado por configurações como fetch.min.bytes (quantidade mínima de dados a ser retornada em uma requisição de busca) e fetch.max.wait.ms (tempo máximo que o broker espera antes de responder a uma requisição de busca se fetch.min.bytes não for atingido). O parâmetro max.poll.records controla o número máximo de registros retornados em uma única chamada poll().19 O processamento em lote no lado do consumidor também pode melhorar a eficiência.
4. **Compressão:** Os produtores podem comprimir os lotes de mensagens antes de enviá-los aos brokers usando algoritmos como Gzip, Snappy, LZ4 ou ZSTD.19 A compressão reduz o tamanho dos dados transmitidos pela rede e armazenados em disco, o que pode aumentar a vazão efetiva e reduzir os custos de armazenamento, especialmente para dados textuais ou repetitivos. Os brokers armazenam os dados comprimidos e os consumidores os descomprimem após recebê-los.
5. **Particionamento e Paralelismo:** A divisão de tópicos em partições permite que a escrita e a leitura ocorram em paralelo. Múltiplos produtores podem escrever para diferentes partições e múltiplos consumidores (em um grupo) podem ler de diferentes partições simultaneamente, aumentando significativamente a capacidade de processamento do sistema.19
6. **Otimizações de Rede e Threads:** Configurações como num.network.threads (threads para lidar com comunicação de rede) e num.io.threads (threads para lidar com I/O de disco) nos brokers podem ser ajustadas para otimizar o uso de recursos de hardware e lidar com um grande número de conexões e requisições concorrentes.19

Embora otimizar para alta vazão possa, por vezes, aumentar a latência (por exemplo, ao aumentar linger.ms ou batch.size), o Kafka oferece as ferramentas para encontrar um equilíbrio adequado para as necessidades específicas da aplicação.19

### **4.2. Durabilidade e Tolerância a Falhas: Replication Factor e min.insync.replicas**

A durabilidade dos dados (garantia de que as mensagens não serão perdidas) e a tolerância a falhas (capacidade do sistema de continuar operando apesar da falha de componentes) são aspectos cruciais do Kafka, alcançados principalmente através da replicação de partições.18

* **Fator de Replicação (Replication Factor):** Para cada partição de um tópico, o Kafka mantém um certo número de cópias, chamadas réplicas. O número total de réplicas é definido pelo replication.factor do tópico.18 Por exemplo, um  
  replication.factor de 3 significa que haverá três cópias de cada partição: uma líder e duas seguidoras.
  + **Líder (Leader):** Todas as operações de escrita (produção) e leitura (consumo) para uma partição passam pela réplica líder.18
  + **Seguidoras (Followers):** As réplicas seguidoras replicam passivamente os dados da líder. Elas não servem requisições de leitura ou escrita diretamente, mas atuam como backups.18  
      
    Se a réplica líder de uma partição falhar, uma das réplicas seguidoras que está sincronizada (in-sync) com a líder é eleita como a nova líder, garantindo a continuidade do serviço.12 Um  
    replication.factor maior aumenta a durabilidade e a disponibilidade dos dados, mas também aumenta o uso de armazenamento e a sobrecarga de replicação na rede.23 Para ambientes de produção, um  
    replication.factor de 3 é comumente recomendado, permitindo a tolerância à falha de até dois brokers (se min.insync.replicas estiver configurado adequadamente) sem perda de dados e um broker sem indisponibilidade.23
* **Réplicas Sincronizadas (In-Sync Replicas - ISR):** Uma réplica seguidora é considerada "in-sync" se estiver acompanhando ativamente a líder, ou seja, se não estiver muito atrasada na replicação dos dados. O conjunto de réplicas (incluindo a líder) que estão sincronizadas é chamado de ISR.
* **Configuração min.insync.replicas:** Esta configuração, definida no nível do broker ou do tópico, especifica o número mínimo de réplicas no ISR (incluindo a líder) que devem ter recebido e confirmado uma mensagem antes que o produtor receba uma confirmação de escrita bem-sucedida. Isso só se aplica quando o produtor está configurado com acks=all (ou acks=-1).18
  + Por exemplo, se replication.factor=3 e min.insync.replicas=2, e o produtor usa acks=all, uma escrita só será considerada bem-sucedida quando a líder e pelo menos uma outra réplica seguidora tiverem confirmado o recebimento da mensagem. Isso garante que a mensagem foi escrita em pelo menos duas réplicas antes de ser confirmada ao produtor.
  + Se o número de réplicas no ISR cair abaixo de min.insync.replicas, o broker líder deixará de aceitar escritas com acks=all para essa partição, gerando uma exceção NotEnoughReplicas ou NotEnoughReplicasAfterAppendException, para evitar a perda de dados.

Trade-off entre Durabilidade e Latência:

A combinação de replication.factor, min.insync.replicas e a configuração acks do produtor permite ajustar o equilíbrio entre durabilidade e latência:

* **Maior Durabilidade (e potencialmente maior latência):**
  + acks=all (ou -1): O produtor espera a confirmação da líder e de todas as ISRs (ou pelo menos min.insync.replicas). Garante que a mensagem foi escrita em múltiplas máquinas antes de ser considerada bem-sucedida.
  + min.insync.replicas próximo ao replication.factor (ex: replication.factor=3, min.insync.replicas=2 ou 3).
* **Menor Latência (e potencialmente menor durabilidade):**
  + acks=1: O produtor espera apenas a confirmação da líder. A mensagem é confirmada mais rapidamente, mas se a líder falhar antes que as seguidoras repliquem, a mensagem pode ser perdida.
  + acks=0: O produtor não espera nenhuma confirmação. Oferece a menor latência e a maior vazão, mas com o menor nível de garantia de entrega (risco de perda de dados).
  + min.insync.replicas=1: Permite que as escritas prossigam mesmo que apenas a líder esteja disponível, aumentando o risco de perda de dados se a líder falhar.

Para a maioria dos ambientes de produção que exigem alta durabilidade, a recomendação é usar replication.factor=3, min.insync.replicas=2 e acks=all nos produtores.18 Isso garante que os dados sejam escritos em pelo menos dois brokers antes da confirmação, tolerando a falha de um broker sem perda de dados.

### **4.3. Escalabilidade (Horizontal)**

A escalabilidade é uma das características mais importantes do Apache Kafka, permitindo que o sistema lide com volumes crescentes de dados e tráfego sem degradação significativa de desempenho.24 O Kafka alcança escalabilidade principalmente através do

**escalonamento horizontal** (scale-out), que envolve adicionar mais nós (brokers) ao cluster.24

**Processo de Adicionar Novos Brokers e Impacto:**

1. **Provisionamento e Configuração de Novos Brokers:**
   * Novos servidores (físicos ou virtuais) são provisionados com os requisitos de hardware adequados (CPU, memória, disco, rede).25
   * O software Kafka é instalado nessas novas máquinas.
   * Os arquivos de configuração dos novos brokers (ex: server.properties) são atualizados para se juntarem ao cluster existente, especificando o mesmo cluster.id (no modo KRaft) ou os endereços do ZooKeeper (no modo ZooKeeper), e um broker.id único para cada novo broker.25
2. **Início dos Novos Brokers:**
   * Quando os novos brokers são iniciados, eles se registram no cluster (com os controladores KRaft ou ZooKeeper) e se tornam membros ativos.25 Inicialmente, esses novos brokers não possuem nenhuma partição de dados.
3. **Redistribuição de Partições (Rebalanceamento):**
   * Para que os novos brokers participem efetivamente do armazenamento e processamento de dados, as partições dos tópicos existentes precisam ser redistribuídas para incluir esses novos brokers. Isso não acontece automaticamente para partições *existentes* apenas por adicionar brokers.
   * A ferramenta kafka-reassign-partitions.sh é usada para gerar um plano de reatribuição de partições e executá-lo. Esse plano especifica como as réplicas das partições serão movidas dos brokers antigos para os novos, ou distribuídas de forma mais equilibrada entre todos os brokers.23
   * O processo de reatribuição de partições move dados pela rede, o que pode consumir recursos. Deve ser feito com cuidado, preferencialmente em horários de menor carga, para minimizar o impacto na performance do cluster.25
   * Para *novos tópicos* criados após a adição dos brokers, o Kafka pode automaticamente distribuir as partições entre todos os brokers disponíveis, incluindo os novos.

**Impacto na Capacidade de Processamento e Armazenamento:**

* **Aumento da Capacidade de Processamento (Throughput):**
  + Com mais brokers, a carga de trabalho de servir requisições de produtores e consumidores é distribuída entre um número maior de máquinas.24
  + Cada broker adiciona sua própria capacidade de CPU, memória e rede ao cluster.
  + A redistribuição de partições permite que mais partições sejam líderes em diferentes brokers, possibilitando maior paralelismo no processamento de leituras e escritas. Isso resulta em maior vazão geral e menor latência.24
* **Aumento da Capacidade de Armazenamento:**
  + Cada novo broker contribui com sua própria capacidade de armazenamento em disco para o cluster.24
  + Com a redistribuição das partições, os dados dos tópicos (e suas réplicas) são espalhados por mais discos, aumentando a capacidade total de armazenamento do cluster. Isso é crucial para lidar com o crescimento do volume de dados e manter políticas de retenção mais longas.24
* **Melhoria da Tolerância a Falhas e Disponibilidade:**
  + Mais brokers significam que o cluster pode tolerar a falha de um número maior de brokers (dependendo do replication.factor) sem perda de dados ou indisponibilidade do serviço.24
  + A replicação de partições entre mais brokers aumenta a redundância dos dados.25
* **Elasticidade:**
  + A capacidade de adicionar (ou remover) brokers permite que o cluster Kafka se adapte dinamicamente a cargas de trabalho variáveis. Recursos podem ser adicionados durante picos de demanda e, teoricamente, reduzidos durante períodos de baixa atividade, otimizando custos (embora a remoção de brokers também exija reatribuição de partições).25

A escalabilidade horizontal é uma das razões pelas quais o Kafka é adequado para lidar com fluxos de dados massivos em tempo real, suportando desde pequenas implantações até clusters com centenas de brokers processando trilhões de mensagens por dia.1

## **5. Ecossistema Kafka**

Além do core do Kafka (brokers, produtores, consumidores), existe um rico ecossistema de ferramentas e bibliotecas que estendem suas funcionalidades e facilitam sua integração e uso em diversos cenários. Três componentes proeminentes são Kafka Connect, Kafka Streams e Schema Registry.

### **5.1. Kafka Connect**

Definição e Propósito:

O Kafka Connect é um framework e uma ferramenta para construir e executar pipelines de dados entre o Apache Kafka e outros sistemas de dados de forma escalável e confiável.2 Ele simplifica o processo de streaming de dados

*para dentro* do Kafka a partir de fontes externas (como bancos de dados, sistemas de mensageria, aplicações) e *para fora* do Kafka para sistemas de destino (como data warehouses, motores de busca, sistemas de análise).26 O Kafka Connect elimina a necessidade de escrever código customizado para tarefas comuns de integração de dados, utilizando componentes reutilizáveis chamados

**conectores**.26

Como Complementa o Core do Kafka:

O Kafka Connect atua como uma ponte robusta entre o Kafka e o restante da infraestrutura de dados de uma organização. Enquanto o core do Kafka foca no transporte e armazenamento de fluxos de eventos, o Kafka Connect lida com a ingestão e egressão desses fluxos, tornando o Kafka um hub central para dados em movimento.

**Principais Características e Componentes:**

* **Conectores (Connectors):** São plugins que implementam a lógica para ler dados de um sistema de origem (Source Connectors) ou escrever dados para um sistema de destino (Sink Connectors).26 Muitos conectores para sistemas populares (JDBC, S3, Elasticsearch, HDFS, MongoDB, etc.) estão disponíveis de fontes abertas ou comerciais.2
* **Tarefas (Tasks):** Um conector pode ser dividido em múltiplas tarefas para paralelizar o trabalho de movimentação de dados.26
* **Workers:** São os processos que executam os conectores e suas tarefas. O Kafka Connect pode rodar em dois modos 26:
  + **Standalone Mode:** Um único worker executa todos os conectores e tarefas. Mais simples de configurar, mas com escalabilidade e tolerância a falhas limitadas. Adequado para desenvolvimento ou pequenas implantações.
  + **Distributed Mode:** Múltiplos workers rodam em diferentes servidores, formando um cluster Connect. Oferece alta escalabilidade, tolerância a falhas e gerenciamento centralizado de configurações via tópicos Kafka. É o modo recomendado para produção.
* **Conversores (Converters):** Usados para serializar e desserializar dados entre o formato do sistema externo e o formato esperado pelo Kafka (e vice-versa).
* **Transformações (Single Message Transforms - SMTs):** Permitem modificações leves nas mensagens individuais à medida que passam pelo pipeline do Connect (ex: filtrar, renomear campos, mascarar dados) sem a necessidade de um sistema de processamento de stream separado.26
* **REST API:** O modo distribuído oferece uma REST API para gerenciar conectores (criar, deletar, pausar, resumir, verificar status).26

**Casos de Uso Comuns:**

* **Integração de Dados e Pipelines de ETL:** Mover dados de bancos de dados relacionais para o Kafka para processamento em tempo real ou carregamento em data lakes/warehouses.26
* **Change Data Capture (CDC):** Capturar alterações em bancos de dados (inserts, updates, deletes) em tempo real e publicá-las como eventos no Kafka.26
* **Agregação de Logs:** Coletar logs de múltiplas fontes e enviá-los para o Kafka para centralização e análise.26
* **Migração para a Nuvem e Soluções Híbridas:** Conectar sistemas on-premises com serviços na nuvem ou integrar dados entre diferentes provedores de nuvem.26
* **Alimentação de Sistemas de Análise em Tempo Real:** Enviar dados do Kafka para plataformas de análise, dashboards ou sistemas de monitoramento.26

### **5.2. Kafka Streams**

Definição e Propósito:

Kafka Streams é uma biblioteca cliente, escrita em Java, para construir aplicações e microsserviços de processamento de fluxos onde os dados de entrada e saída são armazenados em clusters Kafka.2 Ela permite que desenvolvedores escrevam lógica de processamento que consome dados de tópicos Kafka, transforma ou enriquece esses dados e produz os resultados para outros tópicos Kafka, ou interage com armazenamentos de estado externos.2 Diferentemente de outros frameworks de processamento de fluxos que requerem clusters de processamento dedicados (como Apache Flink ou Spark Streaming), uma aplicação Kafka Streams é simplesmente uma aplicação Java/Scala que pode ser executada e escalada como qualquer outra aplicação.29

Como se Diferencia do Kafka Core e Complementa o Ecossistema:

O Kafka Core (brokers) é responsável pelo transporte e armazenamento de fluxos de eventos. Kafka Streams, por sua vez, é uma biblioteca que utiliza o Kafka Core para construir aplicações que agem sobre esses fluxos de eventos em tempo real.29 Ela não requer um cluster de processamento separado; a própria aplicação Streams realiza o processamento.

**Principais Características e APIs:**

* **Sem Cluster Dedicado:** As aplicações Streams são leves e podem ser implantadas em contêineres, VMs ou bare metal, escalando horizontalmente simplesmente executando mais instâncias da aplicação.29
* **Processamento Estado (Stateful Processing):** Suporta operações com estado, como agregações (contagens, somas), junções (joins) entre fluxos e janelamento (windowing) de dados. O estado local é geralmente mantido em um armazenamento RocksDB embutido, que pode persistir em disco e é tolerante a falhas através de changelogs enviados para tópicos Kafka.2
* **Exatamente Uma Vez (Exactly-Once Semantics - EOS):** Oferece garantias de processamento exatamente uma vez, assegurando que cada registro seja processado sem perda ou duplicação, mesmo em caso de falhas.
* **APIs de Alto e Baixo Nível:**
  + **Streams DSL (Domain-Specific Language):** Uma API declarativa de alto nível com operadores funcionais como map, filter, join, groupBy, aggregate, windowedBy, facilitando a escrita de transformações de fluxo comuns.2
  + **Processor API:** Uma API imperativa de baixo nível que oferece controle mais granular sobre o processamento de mensagens e o gerenciamento de estado, permitindo a implementação de operadores customizados.2
* **Consultas Interativas (Interactive Queries):** Permite que outras aplicações consultem o estado mantido por uma aplicação Kafka Streams, expondo os resultados de agregações ou junções em tempo real via APIs REST, por exemplo.29

**Casos de Uso Comuns:**

* **Análise em Tempo Real:** Calcular métricas, detectar anomalias, gerar alertas a partir de fluxos de dados contínuos (ex: detecção de fraude em transações financeiras).29
* **Transformação e Enriquecimento de Dados:** Limpar, normalizar, transformar formatos de dados ou enriquecer eventos com informações de outras fontes em tempo real.
* **Microsserviços Orientados a Eventos:** Construir microsserviços que reagem a eventos, processam-nos e produzem novos eventos.
* **Aplicações de IoT:** Processar dados de sensores, agregar informações e acionar ações com base em padrões identificados.
* **Alimentação de Data Lakehouses:** Processar e transformar dados em tempo real antes de carregá-los em arquiteturas de data lakehouse para análises mais complexas.29

### **5.3. Confluent Schema Registry**

Definição e Propósito:

O Confluent Schema Registry é um serviço centralizado que fornece uma camada para gerenciamento e validação de esquemas de mensagens no Kafka.31 Um esquema define a estrutura dos dados (campos, tipos de dados). O Schema Registry armazena versões de esquemas e permite que produtores e consumidores garantam que os dados que estão sendo trocados são compatíveis com um esquema esperado.31 Ele suporta formatos de serialização como Apache Avro (mais comumente usado), JSON Schema e Protobuf.32

Como Ajuda na Governança de Dados e Evolução de Esquemas:

A governança de dados é crucial em pipelines de dados, especialmente quando múltiplos produtores e consumidores interagem com os mesmos tópicos Kafka. O Schema Registry desempenha um papel vital:

1. **Imposição de Esquemas e Qualidade dos Dados:**
   * Produtores registram o esquema dos dados que estão enviando para um tópico.
   * Consumidores podem buscar o esquema do Registry para desserializar e validar os dados que recebem.
   * Isso garante que os dados que fluem pelo Kafka estejam estruturados e em conformidade com um contrato definido, prevenindo erros de desserialização e garantindo a qualidade dos dados.33
2. **Evolução de Esquemas e Compatibilidade:**
   * Os esquemas de dados inevitavelmente mudam ao longo do tempo (novos campos são adicionados, campos são removidos ou modificados). O Schema Registry gerencia a evolução desses esquemas.
   * Ele permite definir regras de compatibilidade para cada tópico (ou "assunto", que geralmente corresponde a um tópico). Tipos comuns de compatibilidade incluem:
     + BACKWARD: Consumidores usando o novo esquema podem ler dados produzidos com o esquema antigo. (Ex: adicionar um campo opcional).
     + FORWARD: Consumidores usando o esquema antigo podem ler dados produzidos com o novo esquema. (Ex: remover um campo opcional).
     + FULL: O novo esquema é compatível tanto para trás quanto para frente.
     + NONE: Nenhuma verificação de compatibilidade é feita.
   * Quando um produtor tenta registrar uma nova versão de um esquema, o Registry verifica se a mudança é compatível com as versões anteriores, de acordo com a regra definida. Isso previne que mudanças incompatíveis quebrem os consumidores existentes.33
3. **Repositório Centralizado de Metadados:**
   * Atua como uma fonte única de verdade para a estrutura dos dados, facilitando o entendimento e a colaboração entre diferentes equipes que produzem ou consomem dados do Kafka.31
4. **Serialização/Desserialização Eficiente com Avro:**
   * O Apache Avro é um formato de serialização binário compacto e eficiente, que armazena o esquema junto com os dados (ou uma referência a ele).
   * Quando usado com o Schema Registry, os produtores serializam os dados com Avro e, em vez de enviar o esquema completo com cada mensagem, enviam apenas um ID do esquema.
   * Os consumidores recebem o ID do esquema, buscam o esquema correspondente no Registry (se ainda não o tiverem em cache) e usam-no para desserializar os dados. Isso reduz significativamente o tamanho das mensagens e a sobrecarga de rede.

Como Complementa o Core do Kafka:

Enquanto o Kafka Core lida com o transporte de bytes, o Schema Registry adiciona uma camada de semântica e governança sobre esses bytes, garantindo que eles possam ser interpretados corretamente e que as mudanças na estrutura dos dados não causem interrupções nos pipelines. Ele é essencial para construir pipelines de dados robustos, evolutivos e fáceis de manter em escala.

Em conjunto, Kafka Connect, Kafka Streams e Schema Registry formam um ecossistema poderoso que permite não apenas mover dados, mas também processá-los e governá-los de forma eficaz em tempo real.

## **6. Casos de Uso Comuns do Apache Kafka**

A flexibilidade, escalabilidade e desempenho do Apache Kafka o tornam adequado para uma ampla gama de casos de uso em diversas indústrias. Alguns dos mais proeminentes incluem:

1. Pipelines de Dados em Tempo Real:  
   O Kafka é frequentemente o coração de pipelines de dados em tempo real, atuando como um buffer confiável e escalável entre sistemas produtores e consumidores de dados.1 Ele pode ingerir dados de várias fontes (bancos de dados, aplicações, sensores IoT, logs) e disponibilizá-los para processamento, análise ou armazenamento em sistemas downstream em tempo real ou batch.5 Isso é crucial para empresas que precisam tomar decisões rápidas baseadas nos dados mais recentes.5 O Kafka Connect é uma ferramenta chave para construir esses pipelines, facilitando a entrada e saída de dados do Kafka.28
2. Agregação de Logs:  
   Em sistemas distribuídos e arquiteturas de microsserviços, coletar e analisar logs de centenas ou milhares de fontes pode ser um desafio. O Kafka serve como um destino centralizado e de alta capacidade para agregar logs de aplicações, servidores, bancos de dados e outros componentes do sistema.3 Uma vez no Kafka, os logs podem ser consumidos por sistemas de análise de logs (como Elasticsearch, Splunk), ferramentas de monitoramento ou sistemas de alerta.3 A durabilidade e a capacidade de retenção do Kafka garantem que os logs não sejam perdidos e possam ser reprocessados se necessário. Empresas como Netflix e Microsoft Azure utilizam Kafka para agregação de logs em larga escala.35
   * **Arquitetura de Referência para Agregação de Logs:** Uma arquitetura comum envolve:
     + **Agentes de Coleta (Ingestão):** Ferramentas como Fluentd, Logstash ou Filebeat são instaladas nos servidores de origem para coletar logs e enviá-los para um tópico Kafka.35 O Kafka Connect também pode ser usado com conectores específicos.
     + **Apache Kafka:** Atua como o buffer central, durável e escalável para os fluxos de logs.
     + **Consumidores/Processadores:**
       - **Kafka Streams/Flink:** Para processamento em tempo real dos logs (filtragem, enriquecimento, detecção de anomalias).
       - **Kafka Connect (Sink Connectors):** Para enviar os logs do Kafka para sistemas de armazenamento e análise.
     + **Armazenamento e Análise (Consumo):**
       - **Elasticsearch:** Um motor de busca e análise popular para logs, frequentemente usado com o Kibana para visualização e dashboards.35
       - Outros sistemas como Splunk, data lakes (HDFS, S3) ou SIEMs (Security Information and Event Management).38  
           
         A Confluent, por exemplo, descreve uma arquitetura de referência onde logs de firewall, IPS, login e web são ingeridos via Kafka Connect, processados com Flink para detecção de anomalias e, em seguida, enviados para Elastic/Kibana e Google BigQuery, com governança de esquemas via Schema Registry.38
3. Event Sourcing:  
   O Kafka é uma excelente escolha para implementar o padrão de arquitetura event sourcing. Nesse padrão, todas as alterações no estado de uma aplicação são capturadas como uma sequência de eventos imutáveis e ordenados.3 O Kafka atua como o  
   *event store* (armazenamento de eventos) durável e de alta performance.3
   * Quando o estado de uma entidade muda (ex: um pedido é criado, um item é adicionado ao carrinho), um evento descrevendo essa mudança é publicado em um tópico Kafka.
   * O estado atual da aplicação pode ser reconstruído a qualquer momento, reproduzindo (replaying) a sequência de eventos desde o início.5
   * Isso fornece um histórico completo de auditoria, facilita a depuração e permite que diferentes projeções do estado sejam criadas para diferentes necessidades de consulta.
4. Coleta de Métricas e Monitoramento:  
   Similar à agregação de logs, o Kafka é usado para coletar métricas de desempenho e operacionais de aplicações distribuídas, servidores, contêineres e infraestrutura de rede.3 Métricas como utilização de CPU/memória, latência de requisições, taxas de erro, contadores de negócios, etc., podem ser publicadas como fluxos de eventos no Kafka.3  
     
   Sistemas de monitoramento (como Prometheus, Grafana, Datadog) e plataformas de alerta podem consumir essas métricas do Kafka para fornecer visibilidade em tempo real sobre a saúde e o desempenho dos sistemas, permitindo a detecção proativa de problemas e a análise de tendências.3
5. Rastreamento de Atividade de Websites (Clickstream Analysis):  
   O Kafka pode rastrear a atividade do usuário em websites e aplicativos móveis em tempo real. Cada ação do usuário (visualização de página, clique, pesquisa, adição ao carrinho, login) pode ser publicada como um evento em um tópico Kafka.3 Esses dados de clickstream podem ser usados para:
   * Personalização em tempo real.
   * Sistemas de recomendação.
   * Análise de funil de conversão.
   * Detecção de fraude.
   * Testes A/B.
6. Processamento de Fluxos (Stream Processing):  
   Com a biblioteca Kafka Streams ou integrando-se com outros motores de processamento de fluxo como Apache Flink ou Apache Spark Streaming, o Kafka permite o processamento complexo de dados em movimento.1 Isso inclui transformações, agregações, junções de fluxos, enriquecimento de dados e aplicação de modelos de machine learning em tempo real.
7. Sistemas de Mensageria:  
   Embora tenha evoluído para uma plataforma de streaming de eventos, o Kafka ainda pode ser usado como um sistema de mensageria tradicional de alta performance, substituindo ou complementando message brokers como RabbitMQ ou ActiveMQ em cenários que exigem maior escalabilidade e durabilidade.3 Ele facilita a comunicação assíncrona e desacoplada entre microsserviços.

A capacidade do Kafka de lidar com grandes volumes de dados, sua baixa latência, durabilidade e escalabilidade o tornam uma tecnologia fundamental para empresas que buscam se tornar orientadas a eventos e dados em tempo real.

## **7. Monitoramento de um Cluster Kafka**

Monitorar um cluster Kafka em produção é essencial para garantir sua saúde, desempenho, disponibilidade e identificar proativamente possíveis problemas. Diversas métricas em diferentes níveis (broker, produtor, consumidor, ZooKeeper/KRaft) precisam ser acompanhadas.

**Métricas Críticas para Monitorar:**

A seguir, algumas das métricas mais importantes, com destaque para 'Consumer Lag' e 'Under-Replicated Partitions':

**Métricas do Broker e Cluster:**

* **Under-Replicated Partitions (kafka.replication.under\_replicated\_partitions):** 39
  + **O que é:** Indica o número de partições que não possuem o número desejado de réplicas sincronizadas (definido pelo replication.factor). Idealmente, este valor deve ser 0.
  + **Importância:** Um valor maior que 0 significa que algumas partições têm menos réplicas ativas do que o configurado, o que reduz a tolerância a falhas e aumenta o risco de perda de dados se mais brokers falharem. Pode ser causado por falha de brokers, problemas de rede ou brokers sobrecarregados que não conseguem replicar dados rapidamente. É um alerta crítico que precisa de investigação imediata.
* **ActiveControllerCount (kafka.controller.active.count):** 39
  + **O que é:** O número de controladores ativos no cluster.
  + **Importância:** Deve ser exatamente 1. Qualquer outro valor indica um problema sério no cluster (ex: múltiplos controladores - "split brain", ou nenhum controlador).
* **OfflinePartitionsCount (kafka.replication.offline\_partitions\_count):** 39
  + **O que é:** Número de partições cujos líderes não estão disponíveis.
  + **Importância:** Partições offline não podem ser escritas nem lidas. Um valor maior que 0 indica indisponibilidade de dados e requer atenção imediata.
* **Taxa de Bytes In/Out (kafka.net.bytes\_in.rate, kafka.net.bytes\_out.rate):** 39
  + **O que é:** Taxa de bytes recebidos e enviados pelo broker.
  + **Importância:** Ajuda a entender a carga de rede no broker e no cluster, identificar gargalos e planejar capacidade.
* **Taxa de Mensagens In (kafka.messages\_in.rate):** 39
  + **O que é:** Número de mensagens recebidas pelo broker por segundo.
  + **Importância:** Indica a taxa de produção de mensagens para o broker.
* **Request Queue Size (kafka.request.channel.queue.size):** 39
  + **O que é:** Tamanho da fila de requisições de rede aguardando para serem processadas.
  + **Importância:** Um valor consistentemente alto pode indicar que o broker está sobrecarregado e não consegue processar as requisições rapidamente.
* **Request Latency (p99, avg - ex: kafka.request.produce.time.99percentile):** 39
  + **O que é:** Latência para processar diferentes tipos de requisições (Produce, FetchConsumer, FetchFollower).
  + **Importância:** Ajuda a identificar gargalos de desempenho no broker. Picos de latência podem afetar produtores e consumidores.
* **Log Flush Rate/Time (kafka.log.flush\_rate.rate):** 39
  + **O que é:** Frequência ou tempo gasto para descarregar dados do cache de página para o disco.
  + **Importância:** Importante para a durabilidade dos dados. Problemas aqui podem indicar I/O de disco lento.
* **ISR Shrinks/Expands Rate (kafka.replication.isr\_shrinks.rate, kafka.replication.isr\_expands.rate):** 39
  + **O que é:** Taxa com que réplicas entram ou saem do conjunto de réplicas sincronizadas (ISR).
  + **Importância:** Flutuações frequentes podem indicar instabilidade nos brokers ou problemas de rede.
* **Leader Election Rate (kafka.replication.leader\_elections.rate):** 39
  + **O que é:** Taxa de eleições de líder.
  + **Importância:** Um aumento súbito geralmente indica falhas de brokers.39

**Métricas do Consumidor:**

* **Consumer Lag (kafka.consumer\_lag):** 8
  + **O que é:** A diferença entre o último offset produzido para uma partição (Log-End Offset) e o último offset commitado por um grupo de consumidores para essa mesma partição. Essencialmente, mede quantos registros um consumidor está "atrasado" em relação ao produtor.
  + **Importância:** É uma das métricas mais críticas para monitorar o desempenho do consumidor. Um lag crescente indica que os consumidores não estão conseguindo processar as mensagens na mesma velocidade em que são produzidas. Isso pode levar a um acúmulo de dados no Kafka, aumento da latência de ponta a ponta e, eventualmente, ao esgotamento do espaço em disco se a retenção for longa. Causas comuns incluem consumidores lentos, processamento intensivo de mensagens, número insuficiente de consumidores ou problemas de rede.
* **Fetch Rate/Size (kafka.consumer.records\_consumed, kafka.consumer.fetch\_size\_avg):** 39
  + **O que é:** Taxa de mensagens consumidas e tamanho médio das buscas.
  + **Importância:** Ajuda a entender a eficiência do consumidor.

**Métricas do Produtor:**

* **Produce Rate (kafka.producer.record\_send\_rate):** 39
  + **O que é:** Taxa de mensagens enviadas pelo produtor.
  + **Importância:** Monitora a atividade do produtor.
* **Produce Error Rate (kafka.producer.record\_error\_rate):** 39
  + **O que é:** Taxa de erros na produção de mensagens.
  + **Importância:** Erros podem indicar problemas de configuração, brokers indisponíveis ou falhas de autorização.
* **Produce Latency (kafka.producer.request\_latency\_avg):** 39
  + **O que é:** Latência média das requisições de produção.
  + **Importância:** Afeta o desempenho da aplicação produtora.

**Métricas do ZooKeeper (se aplicável) / KRaft:**

* **ZooKeeper:** Disponibilidade, latência de requisições, número de conexões ativas, tamanho da fila de requisições pendentes.40
* **KRaft:** Saúde do quórum de controladores, latência de replicação do log de metadados, taxa de eleição de líder do controlador.

**Ferramentas de Monitoramento:**

* **JMX (Java Management Extensions):** O Kafka expõe a maioria de suas métricas via JMX, que é o reporter padrão.40 Ferramentas como JConsole, VisualVM ou coletores JMX (como o JMX Exporter para Prometheus) podem ser usadas.
* **Confluent Control Center / Confluent Cloud Monitoring:** Oferecem dashboards e alertas prontos para uso para monitorar clusters Kafka e componentes da plataforma Confluent.40 O Confluent Metrics Reporter é necessário para coletar métricas para o Control Center e Auto Data Balancer, publicando-as em um tópico Kafka.41
* **Prometheus e Grafana:** Uma combinação popular para coletar métricas (via JMX Exporter ou OpenTelemetry Collector) e visualizá-las em dashboards customizáveis.5
* **Datadog:** Oferece integração para coletar métricas do Kafka, incluindo métricas do OpenTelemetry Collector e JMX.39 O Datadog fornece um dashboard "Kafka, Zookeeper and Kafka Consumer Overview".39
* **Outras Soluções APM (Application Performance Monitoring):** Muitas ferramentas APM possuem integrações para monitorar Kafka.

É crucial configurar alertas para métricas críticas (especialmente Under-Replicated Partitions, OfflinePartitionsCount, ActiveControllerCount e Consumer Lag) para permitir uma resposta rápida a problemas.40 O monitoramento contínuo não apenas ajuda a manter a estabilidade do cluster, mas também fornece insights para otimizações de desempenho e planejamento de capacidade.

## **8. Melhores Práticas**

Para operar e utilizar o Apache Kafka de forma eficiente, segura e escalável, é fundamental seguir um conjunto de melhores práticas que abrangem desde o design de tópicos e partições até a configuração de segurança e otimização de desempenho.

### **8.1. Estratégia de Tópicos e Partições**

A forma como os tópicos são particionados e como as chaves de mensagens são usadas tem um impacto direto no desempenho, escalabilidade e na garantia de ordem das mensagens.

**Escolhendo o Número de Partições:**

* **Paralelismo vs. Sobrecarga:** O número de partições de um tópico determina o nível máximo de paralelismo para os consumidores de um grupo (um consumidor por partição).12 Mais partições podem levar a uma maior vazão, pois o trabalho é distribuído. No entanto, um número excessivo de partições pode aumentar a sobrecarga de gerenciamento nos brokers (mais arquivos abertos, mais metadados) e no ZooKeeper/KRaft, além de potencialmente aumentar a latência de ponta a ponta e o tempo de recuperação em caso de falhas.12
* **Vazão Desejada:** Estime a vazão de produção e consumo por partição. Se uma única partição pode sustentar X MB/s, e você precisa de Y MB/s para o tópico, então Y/X pode ser um ponto de partida para o número de partições.
* **Número de Consumidores:** Considere o número de instâncias de consumidores que você planeja executar em paralelo. O número de partições deve ser, no mínimo, igual ao número de consumidores concorrentes que você deseja para um grupo.
* **Crescimento Futuro:** Planeje para o crescimento futuro. É mais fácil adicionar partições a um tópico (embora isso possa impactar a ordem se chaves forem usadas e a lógica de particionamento mudar) do que reduzir o número de partições (o que não é suportado diretamente).13
* **Balanceamento entre Brokers:** Distribua as partições (e seus líderes) uniformemente entre os brokers para balancear a carga de I/O, CPU e rede.5

Estratégia de Chaveamento (Partition Key):

A chave de uma mensagem Kafka é usada pelo produtor para determinar para qual partição a mensagem será enviada.

* **Garantia de Ordem:** Se a ordem das mensagens é crucial para um subconjunto específico de dados (ex: todas as atualizações para um mesmo cliente devem ser processadas em ordem), use uma chave de partição consistente para esse subconjunto (ex: customer\_id).5 O Kafka garante que todas as mensagens com a mesma chave serão enviadas para a mesma partição e, portanto, consumidas em ordem por um consumidor daquela partição.13
* **Distribuição de Carga (Evitar Hotspots):** A escolha da chave de partição é vital para uma distribuição uniforme dos dados entre as partições. Uma chave mal escolhida pode levar a "hotspots" – partições que recebem um volume desproporcional de dados, tornando-se gargalos.12
  + **Cardinalidade da Chave:** Use chaves com alta cardinalidade (muitos valores únicos) para garantir uma boa distribuição. Chaves com baixa cardinalidade (ex: boolean\_flag) resultarão em poucas partições ativas.
  + **Hashing:** Por padrão, o Kafka usa um hash da chave (Murmur2) para mapeá-la para uma partição: hash(key) % numPartitions.13 Certifique-se de que a chave gere uma gama variada de valores de hash.
* **Sem Chave (Round-Robin):** Se a ordem das mensagens não é uma preocupação e o objetivo principal é a distribuição uniforme da carga, não especificar uma chave fará com que o produtor distribua as mensagens de forma round-robin entre as partições disponíveis (comportamento padrão se a chave for nula).12 Isso geralmente resulta em um bom balanceamento de carga.
* **Particionador Customizado:** Para lógicas de particionamento mais complexas, é possível implementar um particionador customizado (partitioner.class na configuração do produtor).12

**Impacto na Performance e Ordem:**

* **Performance:** Um número adequado de partições e uma boa estratégia de chaveamento distribuem a carga eficientemente, maximizando a vazão e minimizando a latência. Hotspots ou um número inadequado de partições podem degradar o desempenho.
* **Ordem:** A ordem é garantida apenas *dentro de uma partição*. O uso de chaves é o mecanismo para garantir que mensagens relacionadas sejam processadas em sequência. Adicionar partições a um tópico existente pode alterar o mapeamento de chaves para partições (se o número de partições for usado no cálculo do hash), o que pode quebrar as garantias de ordenação para chaves existentes, a menos que uma lógica de particionamento consistente seja mantida.13

**Outras Recomendações:**

* **Monitore Métricas de Partição:** Acompanhe a saúde das partições, status de replicação, tamanho, taxa de crescimento e latência de mensagens para identificar problemas e otimizar a estratégia.12
* **Teste em Staging:** Valide suas estratégias de partição em um ambiente de teste que simule sua carga de produção antes de aplicar em produção.12

### **8.2. Otimização de Desempenho (Tuning) de Produtores e Consumidores**

Ajustar as configurações de produtores e consumidores é crucial para alcançar a vazão e latência desejadas.

Otimização do Produtor:

O objetivo geralmente é aumentar a vazão, o que pode envolver um trade-off com a latência.

* **batch.size (padrão: 16384 bytes):** Aumentar o batch.size (ex: para 100KB-200KB) permite que o produtor agrupe mais dados em cada requisição enviada ao broker. Isso reduz a sobrecarga de rede por mensagem e melhora a eficiência da compressão, resultando em maior vazão.21 Um  
  batch.size maior pode aumentar a latência, pois o produtor espera mais para preencher o lote.
* **linger.ms (padrão: 0 ms):** Define o tempo máximo que o produtor espera para acumular dados para um lote, mesmo que batch.size não seja atingido. Aumentar linger.ms (ex: para 10-100 ms) permite que mais mensagens sejam agrupadas, especialmente em cenários de baixa taxa de produção, melhorando a vazão ao custo de uma pequena latência adicional.19
* **compression.type (padrão: none):** Habilitar compressão (ex: lz4, snappy, gzip, zstd) reduz o tamanho dos dados enviados pela rede e armazenados em disco. Isso pode aumentar significativamente a vazão efetiva, especialmente para dados textuais ou com padrões repetitivos, e reduzir custos de rede e armazenamento.19 A compressão adiciona alguma sobrecarga de CPU no produtor (para comprimir) e no consumidor (para descomprimir), mas o ganho na rede geralmente compensa. LZ4 e Snappy oferecem um bom equilíbrio entre taxa de compressão e uso de CPU.
* **acks (padrão: all desde Kafka 3.0, anteriormente 1):** Controla o nível de reconhecimento esperado do broker.
  + acks=all (ou -1): Maior durabilidade. O líder espera que todas as réplicas no ISR (ou min.insync.replicas) confirmem a escrita. Pode aumentar a latência.
  + acks=1: O líder confirma a escrita assim que a recebe, sem esperar pelas réplicas. Menor latência, mas menor durabilidade (risco de perda se o líder falhar antes da replicação).
  + acks=0: O produtor não espera nenhuma confirmação. Menor latência e maior vazão, mas sem garantia de entrega.  
    Para alta vazão, acks=1 pode ser considerado se uma pequena chance de perda de dados for aceitável. Caso contrário, acks=all com min.insync.replicas bem configurado é o mais seguro.21
* **buffer.memory (padrão: 33554432 bytes - 32MB):** Memória total que o produtor pode usar para bufferizar registros aguardando envio ao servidor. Se essa memória encher, chamadas send() podem bloquear ou gerar exceções.
* **Envio Assíncrono:** Utilizar o envio assíncrono (producer.send() com um callback) permite que a aplicação produtora continue processando enquanto as mensagens são enviadas em background, melhorando o desempenho.22

**Otimização do Consumidor:**

* **fetch.min.bytes (padrão: 1 byte):** Quantidade mínima de dados que o servidor deve retornar para uma requisição de busca. Aumentar este valor pode reduzir a carga no broker, pois ele responde com lotes maiores, mas pode aumentar a latência se os dados não chegarem rapidamente.
* **fetch.max.wait.ms (padrão: 500 ms):** Tempo máximo que o servidor espera antes de responder a uma requisição de busca se fetch.min.bytes não for atingido.
* **max.poll.records (padrão: 500):** Número máximo de registros retornados em uma única chamada poll(). Aumentar este valor pode melhorar a vazão se o consumidor puder processar mais mensagens por vez, mas também pode aumentar o tempo de processamento por chamada poll() e o risco de timeouts de rebalanceamento se o processamento for lento.19
* **max.partition.fetch.bytes (padrão: 1MB):** Tamanho máximo de dados por partição que o servidor retornará. Deve ser grande o suficiente para acomodar as maiores mensagens.
* **Paralelismo de Consumidores:** Aumentar o número de instâncias de consumidores dentro de um grupo (até o número de partições do tópico) é a principal forma de escalar o consumo.22
* **Configurações de Rebalanceamento de Grupo:**
  + session.timeout.ms (padrão: 10000 ms, aumentado para 45000 ms em versões mais recentes): Tempo máximo que o broker espera por um heartbeat do consumidor antes de considerá-lo morto e iniciar um rebalanceamento.
  + heartbeat.interval.ms (padrão: 3000 ms): Frequência com que o consumidor envia heartbeats. Deve ser menor que session.timeout.ms, geralmente 1/3.
  + max.poll.interval.ms (padrão: 300000 ms - 5 minutos): Tempo máximo entre chamadas poll() antes que o consumidor seja considerado falho. Se o processamento de um lote de mensagens levar mais tempo que isso, o consumidor pode ser removido do grupo. Ajuste com base no tempo de processamento das suas mensagens.

**Geral (Broker, Tópico, SO, JVM):**

* **Número de Partições:** Como discutido anteriormente, impacta diretamente o paralelismo.
* **Configurações de I/O do Broker:** num.io.threads (para I/O de disco) e num.network.threads (para I/O de rede).19
* **Tuning do Sistema Operacional:** Aumentar limites de arquivos abertos, ajustar configurações de rede TCP/IP.
* **Tuning da JVM:** Escolher o coletor de lixo (Garbage Collector) apropriado (ex: G1GC para baixa latência, ParallelGC para alta vazão) e ajustar o tamanho do heap.22
* **TLS/SSL:** A criptografia adiciona sobrecarga. Otimizações de hardware (CPUs com AES-NI) podem ajudar.

A otimização de desempenho é um processo iterativo. É crucial monitorar as métricas chave (vazão, latência, consumer lag, uso de CPU/memória) e testar diferentes combinações de configurações em um ambiente representativo para encontrar os valores ideais para a carga de trabalho específica.5

### **8.3. Segurança: Criptografia (TLS/SSL), Autenticação (SASL) e Autorização (ACLs)**

Proteger um cluster Kafka é fundamental, especialmente quando lida com dados sensíveis. A segurança no Kafka é construída sobre três pilares principais:

1. **Criptografia em Trânsito (TLS/SSL):**
   * **Propósito:** Garante que os dados trocados entre clientes (produtores/consumidores) e brokers, bem como entre os próprios brokers (comunicação inter-broker) e entre brokers e ZooKeeper/KRaft controllers, sejam criptografados e protegidos contra interceptação (escuta) e adulteração.5
   * **Como Funciona:** O TLS (Transport Layer Security), sucessor do SSL (Secure Sockets Layer), usa criptografia de chave pública/privada para estabelecer um canal de comunicação seguro. Certificados digitais são usados para verificar a identidade dos servidores (e opcionalmente dos clientes, para TLS mútuo). Uma vez que o handshake TLS é concluído, os dados são criptografados usando chaves simétricas negociadas.
   * **Configuração:** Requer a configuração de keystores e truststores nos brokers e clientes, especificando os certificados e chaves privadas. Os listeners nos brokers devem ser configurados para usar protocolos como SSL ou SASL\_SSL.
2. **Autenticação (SASL - Simple Authentication and Security Layer):**
   * **Propósito:** Verifica a identidade dos clientes (produtores, consumidores, ferramentas administrativas) e brokers que tentam se conectar ao cluster.43 Antes de autorizar qualquer ação, o Kafka precisa saber  
     *quem* está fazendo a requisição.
   * **Como Funciona:** SASL é um framework que permite diferentes mecanismos de autenticação. O cliente e o servidor negociam um mecanismo comum.
   * **Mecanismos SASL Suportados pelo Kafka:**
     + **SASL/PLAIN:** Autenticação simples baseada em nome de usuário e senha. As credenciais são enviadas em texto plano (da perspectiva do SASL), portanto, **deve sempre ser usado sobre uma conexão TLS (SASL\_SSL)** para proteger as credenciais em trânsito.43
     + **SASL/SCRAM (SCRAM-SHA-256, SCRAM-SHA-512):** Salted Challenge Response Authentication Mechanism. Um mecanismo mais seguro de nome de usuário/senha que usa desafios e não envia a senha diretamente, protegendo contra sniffing e ataques de dicionário.44 Recomendado sobre SASL/PLAIN para produção.43
     + **SASL/GSSAPI (Kerberos):** Autenticação forte baseada em tickets, integrada com infraestrutura Kerberos existente. Mais complexo de configurar, mas oferece alta segurança.44
     + **SASL/OAUTHBEARER:** Permite autenticação usando tokens OAuth 2.0, útil para integração com sistemas de identidade modernos.
   * **Configuração:** Envolve a configuração de propriedades SASL nos brokers (sasl.enabled.mechanisms, sasl.mechanism.inter.broker.protocol) e nos clientes (sasl.mechanism, security.protocol). Também requer a configuração de arquivos JAAS (Java Authentication and Authorization Service) para definir os módulos de login para cada mecanismo.43
3. **Autorização (ACLs - Access Control Lists):**
   * **Propósito:** Uma vez que um cliente é autenticado, a autorização determina *quais operações* esse cliente (chamado de Principal) tem permissão para realizar em *quais recursos* do Kafka.34
   * **Como Funciona:** As ACLs são regras que concedem (Allow) ou negam (Deny) permissões. Cada ACL especifica:
     + Principal: O usuário autenticado (ex: User:alice).
     + Operation: A ação (ex: Read, Write, Create, Describe, Alter, Delete, All).
     + Resource Type: O tipo de recurso (ex: Topic, Group (para consumer group), Cluster, TransactionalId, DelegationToken).
     + Resource Name: O nome específico do recurso (ex: nome do tópico, nome do grupo). Pode usar wildcards.
     + Host: O endereço IP do qual o principal pode se conectar (padrão: \*, todos os hosts).
     + Permission Type: Allow ou Deny.
   * **Gerenciamento:** As ACLs são gerenciadas usando a ferramenta kafka-acls.sh ou APIs de administração.
   * **Autorizador:** O Kafka usa um componente autorizador plugável (authorizer.class.name). O AclAuthorizer padrão armazena ACLs no ZooKeeper (para modo ZooKeeper) ou o StandardAuthorizer as armazena nos metadados do cluster (para modo KRaft).43
   * **Princípio do Menor Privilégio:** É uma boa prática conceder apenas as permissões estritamente necessárias para cada principal.43 Por exemplo, um produtor precisa de permissão  
     Write e Describe para um tópico, enquanto um consumidor precisa de Read e Describe para o tópico e Read para o grupo de consumidores.

Como Funcionam Juntos:

Os três pilares trabalham em conjunto para fornecer uma segurança em camadas:

1. Um cliente tenta se conectar a um broker.
2. Se o TLS estiver habilitado, uma conexão segura é estabelecida primeiro, criptografando toda a comunicação subsequente.
3. O cliente então se autentica no broker usando o mecanismo SASL configurado. O broker verifica a identidade do cliente.
4. Se a autenticação for bem-sucedida, para cada operação que o cliente tenta realizar (ex: produzir para um tópico, consumir de um tópico), o autorizador do broker verifica as ACLs para determinar se o principal autenticado tem permissão para realizar aquela operação naquele recurso específico a partir daquele host.

A combinação de criptografia, autenticação forte e autorização granular é essencial para proteger os dados e a infraestrutura do Kafka contra acesso não autorizado e ameaças.43 É recomendado sempre usar TLS com SASL em ambientes de produção.43

## **9. Análise Comparativa com Outras Plataformas**

Embora o Apache Kafka seja uma plataforma poderosa para streaming de eventos, é importante entender como ele se compara a outros sistemas de mensageria e filas, como RabbitMQ e AWS SQS, para escolher a ferramenta certa para cada caso de uso.

### **9.1. Kafka vs. RabbitMQ**

**Arquitetura e Modelo de Mensagem:**

* **Kafka:**
  + **Modelo de Log Distribuído:** Kafka é fundamentalmente um log de commit distribuído e particionado. As mensagens são anexadas a partições de tópicos e retidas por um período configurável, independentemente de terem sido consumidas.1
  + **Modelo de Consumo Pull:** Os consumidores "puxam" (pull) mensagens dos brokers Kafka, gerenciando seus próprios offsets para rastrear a posição de leitura.45 Isso permite que os consumidores controlem a taxa de consumo e reprocessem mensagens.
  + **Ordenação:** Garante a ordem das mensagens dentro de cada partição.45
* **RabbitMQ:**
  + **Modelo de Fila Tradicional (Broker de Mensagens):** RabbitMQ é um broker de mensagens tradicional que implementa protocolos como AMQP. As mensagens são publicadas em *exchanges* (trocas) que as roteiam para uma ou mais *queues* (filas) com base em regras de roteamento (bindings e routing keys).45
  + **Modelo de Consumo Push:** Os brokers RabbitMQ geralmente "empurram" (push) mensagens para os consumidores registrados, embora os consumidores possam controlar o fluxo com limites de prefetch.45 As mensagens são normalmente removidas da fila após serem consumidas e confirmadas.
  + **Roteamento Flexível:** Oferece mecanismos de roteamento complexos e flexíveis através de diferentes tipos de exchanges (direct, topic, fanout, headers).45
  + **Prioridade de Mensagens:** Suporta prioridades de mensagens nas filas.45

**Durabilidade e Retenção:**

* **Kafka:** Projetado para alta durabilidade com persistência em disco e replicação entre brokers. A retenção de mensagens é uma característica central, permitindo a releitura e o processamento de fluxos históricos.1
* **RabbitMQ:** As mensagens podem ser persistentes (escritas em disco) ou transitórias (apenas em memória). A durabilidade é garantida para mensagens persistentes em filas duráveis. As Quorum Queues no RabbitMQ oferecem replicação de mensagens entre nós para maior tolerância a falhas, similar à replicação do Kafka.45 Geralmente, as mensagens são descartadas após o consumo.45

**Vazão e Latência:**

* **Kafka:** Otimizado para altíssima vazão (milhões ou bilhões de mensagens por segundo) e baixa latência, especialmente para streaming de grandes volumes de dados.45 A latência pode ser ligeiramente maior devido ao batching e à replicação, mas a vazão é seu ponto forte.
* **RabbitMQ:** Excelente para baixa latência de mensagens individuais e cenários de mensagens transacionais. Sua vazão é geralmente menor que a do Kafka, mas é muito eficaz para distribuição de tarefas e comunicação de microsserviços que exigem respostas rápidas.45

**Escalabilidade:**

* **Kafka:** Projetado para escalabilidade horizontal massiva. Adicionar mais brokers ao cluster aumenta a capacidade de forma relativamente simples.46
* **RabbitMQ:** Suporta clustering e escalabilidade horizontal, mas pode ser mais complexo de gerenciar e escalar para níveis de vazão comparáveis aos do Kafka.46

**Gerenciamento e Complexidade:**

* **Kafka:** Pode ter uma curva de aprendizado mais íngreme e maior complexidade operacional para implantar e gerenciar, especialmente se auto-hospedado.45
* **RabbitMQ:** Geralmente considerado mais fácil de implantar, configurar e gerenciar, especialmente para casos de uso de enfileiramento mais simples.45

**Casos de Uso Ideais:**

* **Kafka:** 45
  + Streaming de eventos em tempo real e análise de grandes volumes de dados (logs, IoT, clickstreams).
  + Pipelines de dados em larga escala.
  + Event sourcing e arquiteturas orientadas a eventos.
  + Sistemas que exigem retenção de mensagens e capacidade de replay.
  + Alta durabilidade e tolerância a falhas em escala.
* **RabbitMQ:** 45
  + Enfileiramento de tarefas e processamento de trabalhos em background.
  + Comunicação entre microsserviços com requisitos de baixa latência e roteamento complexo.
  + Padrões de solicitação-resposta.
  + Cenários que se beneficiam de prioridade de mensagens.
  + Aplicações que precisam de um broker de mensagens mais leve e fácil de gerenciar.

**RabbitMQ Streams:** Vale notar que o RabbitMQ introduziu o RabbitMQ Streams, que adiciona semântica de log (similar ao Kafka) ao RabbitMQ, visando casos de uso de streaming de alta vazão, mantendo a flexibilidade de roteamento do RabbitMQ.45

É possível usar Kafka e RabbitMQ juntos, aproveitando os pontos fortes de cada um. Por exemplo, RabbitMQ para ingestão de mensagens de baixa latência e Kafka para armazenamento durável e processamento de stream em larga escala, com o Kafka Connect atuando como ponte.46

### **9.2. Kafka vs. AWS SQS (Simple Queue Service)**

**Arquitetura e Modelo de Mensagem:**

* **Kafka:**
  + **Modelo de Log Distribuído:** Como descrito anteriormente, é uma plataforma de streaming de eventos baseada em logs particionados.47
  + **Gerenciamento de Offset pelo Consumidor:** Os consumidores Kafka gerenciam seus próprios offsets, dando-lhes controle sobre de onde começam a ler e como lidam com o reprocessamento.47
* **AWS SQS:**
  + **Modelo de Fila Gerenciada:** SQS é um serviço de enfileiramento de mensagens totalmente gerenciado pela AWS. As mensagens são enviadas para uma fila e os consumidores as recuperam.47
  + **Dois Tipos de Filas:**
    - **Standard Queues:** Oferecem máxima vazão, melhor esforço na ordenação (mensagens podem ser entregues fora de ordem) e entrega "pelo menos uma vez".
    - **FIFO (First-In, First-Out) Queues:** Garantem a ordem das mensagens (dentro de um grupo de mensagens) e processamento "exatamente uma vez" (com desduplicação). A vazão é menor que as filas Standard.
  + **Visibilidade Timeout:** No SQS, quando um consumidor lê uma mensagem, ela se torna invisível para outros consumidores por um período configurável (visibility timeout). Se o consumidor não deletar a mensagem dentro desse tempo (após processá-la com sucesso), ela se torna visível novamente para ser processada por outro consumidor. Isso difere do modelo de offset do Kafka.47

**Durabilidade e Retenção:**

* **Kafka:** Alta durabilidade através de persistência em disco e replicação. Retenção de mensagens configurável e longa, permitindo replay.47
* **SQS:** Serviço altamente disponível e durável, com mensagens armazenadas redundantemente em múltiplos servidores AWS. A retenção máxima de mensagens no SQS é de 14 dias.47 Mensagens são geralmente excluídas após o processamento bem-sucedido.

**Vazão e Latência:**

* **Kafka:** Projetado para altíssima vazão e baixa latência em cenários de streaming complexos.47
* **SQS:** Oferece boa vazão e baixa latência, especialmente para enfileiramento simples e desacoplamento de aplicações. Filas Standard têm maior vazão que FIFO.

**Escalabilidade:**

* **Kafka:** Escala horizontalmente adicionando brokers e partições, mas requer gerenciamento manual (se auto-hospedado) ou uso de serviços gerenciados.47
* **SQS:** Escala automaticamente com o volume de mensagens, sendo um serviço totalmente gerenciado pela AWS.47

**Gerenciamento e Complexidade:**

* **Kafka:** Requer configuração, gerenciamento e manutenção (provisionamento, upgrades, segurança) se auto-hospedado, o que implica maior sobrecarga operacional.47 Serviços gerenciados de Kafka (como Confluent Cloud, Amazon MSK) reduzem essa carga.
* **SQS:** Totalmente gerenciado pela AWS, com sobrecarga operacional mínima. A AWS cuida da infraestrutura, escalabilidade, patching e disponibilidade.47 É mais simples de configurar e usar.

**Formato de Dados e Ecossistema:**

* **Kafka:** Suporta qualquer formato de dados (JSON, Avro, Protobuf, binário). Possui um ecossistema rico com Kafka Connect, Kafka Streams, Schema Registry e muitas bibliotecas cliente.47
* **SQS:** Lida principalmente com formatos de texto simples (ex: JSON, XML). Integra-se profundamente com outros serviços AWS (Lambda, S3, EC2, SNS).47

**Custo:**

* **Kafka:** Custos de hardware, armazenamento, rede e pessoal operacional (se auto-hospedado). Serviços gerenciados têm modelos de precificação baseados em instâncias, armazenamento, transferência de dados, etc..48
* **SQS:** Modelo pay-as-you-go, baseado no número de requisições e transferência de dados. Possui um nível gratuito.47

**Casos de Uso Ideais:**

* **Kafka:** 47
  + Streaming de eventos complexos em tempo real e processamento de fluxos.
  + Pipelines de dados de alta vazão e longa retenção.
  + Event sourcing.
  + Agregação de logs e métricas em larga escala.
  + Cenários que exigem replay de mensagens e controle granular sobre o consumo.
* **AWS SQS:** 47
  + Desacoplamento de microsserviços e aplicações distribuídas, especialmente dentro do ecossistema AWS.
  + Filas de tarefas simples e processamento de trabalhos em background.
  + Aplicações serverless (integração com AWS Lambda).
  + Cenários que necessitam de um serviço de enfileiramento totalmente gerenciado com configuração mínima.

Em resumo, Kafka é uma plataforma de streaming poderosa e flexível, ideal para cenários de dados complexos e de alta vazão que exigem retenção e processamento de fluxos. RabbitMQ é um broker de mensagens versátil, ótimo para mensageria tradicional, roteamento complexo e tarefas de baixa latência. SQS é uma solução de enfileiramento gerenciada, simples e eficaz para desacoplar aplicações, principalmente no ambiente AWS. A escolha depende dos requisitos específicos de vazão, latência, durabilidade, complexidade de processamento, modelo de mensagem e sobrecarga de gerenciamento.

## **10. Sessão Prática: Configuração e Exemplos Básicos**

Para consolidar o aprendizado, esta seção apresenta um guia prático para configurar um ambiente Kafka localmente usando Docker e exemplos básicos de código para produzir e consumir mensagens em Python.

### **10.1. Configuração Local com Docker (Modo KRaft)**

Rodar o Kafka localmente com Docker é uma forma rápida de iniciar o desenvolvimento e testes. Utilizaremos o modo KRaft para uma configuração mais simples, sem a dependência do ZooKeeper.

**Pré-requisitos:**

* Docker e Docker Compose instalados e em execução.27

**Arquivo docker-compose.yml para um Cluster Kafka Single-Node em Modo KRaft:**

A Confluent fornece exemplos de arquivos docker-compose.yml que facilitam a inicialização de componentes da plataforma, incluindo o Kafka em modo KRaft. Um exemplo simplificado para um único broker Kafka rodando em modo KRaft pode ser derivado das configurações encontradas nos repositórios da Confluent.27

O tutorial "How to run Kafka locally with Docker" da Confluent Developer 49 descreve o uso de um

docker-compose.yml que inicia um único broker Kafka configurado para usar KRaft. Embora o conteúdo exato do arquivo docker-compose.yml específico do tutorial não esteja integralmente nos snippets, podemos inferir sua estrutura e principais configurações com base nas descrições e comandos fornecidos.

Um docker-compose.yml para um broker Kafka em modo KRaft (single-node) normalmente incluiria:

* Definição do serviço do broker.
* Imagem Docker do Kafka (ex: confluentinc/cp-server ou uma imagem oficial do Apache Kafka).
* Mapeamento de portas (ex: 9092 para clientes externos, 29092 para comunicação interna no Docker).
* Variáveis de ambiente para configurar o Kafka em modo KRaft:
  + KAFKA\_NODE\_ID: ID único para o nó.
  + KAFKA\_PROCESS\_ROLES: Definido como broker,controller para um nó único que atua tanto como broker quanto como controlador KRaft.
  + KAFKA\_CONTROLLER\_QUORUM\_VOTERS: Especifica o(s) nó(s) controlador(es) e suas portas. Para um nó único, seria ele mesmo.
  + KAFKA\_LISTENERS e KAFKA\_ADVERTISED\_LISTENERS: Configura como o Kafka escuta conexões de dentro e de fora da rede Docker.
  + KAFKA\_CONTROLLER\_LISTENER\_NAMES.
  + KAFKA\_INTER\_BROKER\_LISTENER\_NAME.
  + Outras configurações como KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR=1 e KAFKA\_TRANSACTION\_STATE\_LOG\_REPLICATION\_FACTOR=1 para um cluster de nó único.

**Exemplo de docker-compose.yml (baseado em princípios de configuração KRaft single-node):**

YAML

version: '3.8'  
services:  
 broker:  
 image: confluentinc/cp-kafka:latest # Ou uma imagem oficial do Apache Kafka com KRaft  
 container\_name: broker  
 ports:  
 - "9092:9092" # Para clientes fora do Docker  
 - "29092:29092" # Para clientes dentro do Docker ou comunicação inter-broker  
 environment:  
 KAFKA\_NODE\_ID: 1  
 KAFKA\_PROCESS\_ROLES: 'broker,controller'  
 KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: 'CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,PLAINTEXT\_HOST:PLAINTEXT'  
 KAFKA\_LISTENERS: 'PLAINTEXT://:29092,CONTROLLER://:9093,PLAINTEXT\_HOST://:9092'  
 KAFKA\_ADVERTISED\_LISTENERS: 'PLAINTEXT://broker:29092,PLAINTEXT\_HOST://localhost:9092'  
 KAFKA\_CONTROLLER\_QUORUM\_VOTERS: '1@broker:9093' # Nó 1 no host 'broker' na porta 9093  
 KAFKA\_CONTROLLER\_LISTENER\_NAMES: 'CONTROLLER'  
 KAFKA\_INTER\_BROKER\_LISTENER\_NAME: 'PLAINTEXT'  
 KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: '1'  
 KAFKA\_GROUP\_INITIAL\_REBALANCE\_DELAY\_MS: '0'  
 KAFKA\_TRANSACTION\_STATE\_LOG\_REPLICATION\_FACTOR: '1'  
 KAFKA\_TRANSACTION\_STATE\_LOG\_MIN\_ISR: '1'  
 KAFKA\_LOG\_DIRS: '/tmp/kraft-combined-logs'  
 # Geração de UUID do Cluster para KRaft  
 CLUSTER\_ID: 'MkU3OEVBNTcwNTJENDM2Qk' # Substitua por um UUID gerado (ex: $(bin/kafka-storage.sh random-uuid))  
 volumes:  
 -./kraft-data:/tmp/kraft-combined-logs # Persistência de dados (opcional para testes rápidos)  
  
# Para gerar um CLUSTER\_ID, você pode usar o script kafka-storage.sh:  
# docker run --rm confluentinc/cp-kafka:latest kafka-storage.sh random-uuid  
# E então formatar os diretórios de log antes do primeiro 'docker-compose up':  
# docker run --rm -v $(pwd)/kraft-data:/tmp/kraft-combined-logs confluentinc/cp-kafka:latest kafka-storage.sh format -t <CLUSTER\_ID> -c /etc/kafka/kafka.properties --ignore-formatted  
# (Nota: o kafka.properties dentro da imagem pode precisar ser ajustado ou montado se não for o padrão KRaft)

Nota: A configuração exata, especialmente para KAFKA\_LISTENERS e KAFKA\_ADVERTISED\_LISTENERS, é crucial e pode variar dependendo da imagem Docker específica e da versão do Kafka. O CLUSTER\_ID deve ser gerado e os diretórios de log formatados antes da primeira inicialização com KRaft.5

**Passos para Executar:**

1. Salve o conteúdo acima como docker-compose.yml em um diretório.
2. Gere um CLUSTER\_ID (se não estiver hardcoded e se a imagem não o gerar automaticamente). Por exemplo, usando o script kafka-storage.sh de uma imagem Kafka:  
   export KAFKA\_CLUSTER\_ID="$(docker run --rm confluentinc/cp-kafka kafka-storage.sh random-uuid)"  
   echo $KAFKA\_CLUSTER\_ID (e use este valor no docker-compose.yml).
3. Formate os diretórios de log (se necessário pela imagem/configuração):  
   docker run --rm -v $(pwd)/kraft-data:/tmp/kraft-combined-logs confluentinc/cp-kafka kafka-storage.sh format -t $KAFKA\_CLUSTER\_ID -c /etc/kafka/kraft/broker.properties (o caminho do arquivo de config pode variar).
4. Inicie o container Kafka:  
   docker compose up -d 49
5. Verifique se o container está em execução:  
   docker compose ps 27

**Criando um Tópico, Produzindo e Consumindo Mensagens (via console Kafka dentro do container):**

1. Acesse o shell do container do broker:  
   docker exec -it broker /bin/bash (ou sh dependendo da imagem) 49
2. Navegue até o diretório bin do Kafka (ex: /opt/kafka/bin ou /usr/bin).
3. Crie um tópico:  
   ./kafka-topics.sh --create --topic meu-topico --bootstrap-server broker:29092 --partitions 1 --replication-factor 1 49  
     
   (Para comunicação de dentro do container, use broker:29092 ou o listener interno configurado).
4. Inicie um produtor de console:  
   ./kafka-console-producer.sh --topic meu-topico --bootstrap-server broker:29092 49  
     
   Digite algumas mensagens, cada uma seguida por Enter. Pressione Ctrl+C para sair.
5. Inicie um consumidor de console em outro terminal (ou após sair do produtor):  
   ./kafka-console-consumer.sh --topic meu-topico --from-beginning --bootstrap-server broker:29092 49  
     
   Você deverá ver as mensagens produzidas. Pressione Ctrl+C para sair.
6. Para parar e remover os containers:  
   docker compose down

Este setup provê um ambiente Kafka funcional para desenvolvimento e aprendizado.

### **10.2. Exemplos Básicos de Produtor e Consumidor em Python (usando confluent-kafka-python)**

A biblioteca confluent-kafka-python é um wrapper Python popular para a biblioteca C librdkafka, oferecendo alta performance.

Instalação:

pip install confluent-kafka

**Exemplo de Código do Produtor (producer.py):**

Python

from confluent\_kafka import Producer  
import socket  
  
conf = {  
 'bootstrap.servers': 'localhost:9092', # Endereço do broker Kafka  
 'client.id': socket.gethostname()  
}  
  
producer = Producer(conf)  
  
topic = 'meu-topico-python'  
  
def acked(err, msg):  
 """ Callback de entrega invocado quando uma mensagem é entregue com sucesso ou falha permanentemente. """  
 if err is not None:  
 print(f"Falha ao entregar mensagem: {err}")  
 else:  
 print(f"Mensagem entregue a {msg.topic()} [{msg.partition()}] @ offset {msg.offset()}")  
  
for i in range(5):  
 mensagem = f"Olá Kafka de Python! Mensagem {i}"  
 # Produz a mensagem de forma assíncrona  
 # O callback 'acked' será chamado quando a entrega for confirmada ou falhar.  
 producer.produce(topic, key=str(i), value=mensagem, callback=acked)  
 producer.poll(0) # Serve para disparar callbacks de entrega pendentes  
  
# Espera por todas as mensagens pendentes serem entregues  
# O flush() é bloqueante.  
producer.flush()  
  
print("Todas as mensagens foram produzidas.")

**Explicação do Produtor:**

* **Configuração Essencial:** bootstrap.servers é a configuração chave para conectar ao Kafka.17
* **Producer(conf):** Cria uma instância do produtor.
* **producer.produce(topic, key, value, callback):** Envia uma mensagem para o tópico especificado. É uma operação assíncrona.
  + key é opcional, mas útil para garantir que mensagens com a mesma chave vão para a mesma partição.
  + callback é uma função chamada para reportar o status da entrega.17
* **producer.poll(0):** É chamado no loop para permitir que o produtor processe eventos de entrega (callbacks) sem bloquear. Um timeout de 0 significa que ele retorna imediatamente.17
* **producer.flush():** Garante que todas as mensagens enfileiradas e pendentes sejam enviadas e seus callbacks processados antes que o script termine. É uma chamada bloqueante.17

**Exemplo de Código do Consumidor (consumer.py):**

Python

from confluent\_kafka import Consumer, KafkaError, KafkaException  
  
conf = {  
 'bootstrap.servers': 'localhost:9092', # Endereço do broker Kafka  
 'group.id': 'meu-grupo-consumidor-python', # ID do grupo de consumidores  
 'auto.offset.reset': 'earliest' # Começa a ler do início do tópico se não houver offset commitado  
}  
  
consumer = Consumer(conf)  
  
topic = 'meu-topico-python'  
  
try:  
 consumer.subscribe([topic]) # Inscreve-se no tópico  
  
 print(f"Consumindo mensagens do tópico: {topic}. Pressione Ctrl+C para sair.")  
 while True:  
 # poll() busca por novas mensagens. O timeout (ex: 1.0 segundo) define quanto tempo esperar.  
 msg = consumer.poll(timeout=1.0)  
  
 if msg is None: # Nenhuma mensagem recebida dentro do timeout  
 continue  
 if msg.error():  
 if msg.error().code() == KafkaError.\_PARTITION\_EOF:  
 # Fim da partição, não é um erro real.  
 print(f"%% {msg.topic()} [{msg.partition()}] alcançou o fim no offset {msg.offset()}\n")  
 elif msg.error():  
 raise KafkaException(msg.error())  
 else:  
 # Mensagem recebida com sucesso  
 print(f"Recebida mensagem: key={msg.key().decode('utf-8') if msg.key() else None}, "  
 f"value={msg.value().decode('utf-8')}, "  
 f"topic={msg.topic()}, partition={msg.partition()}, offset={msg.offset()}")  
 # O commit de offset pode ser automático (padrão) ou manual.  
 # Para commit manual: consumer.commit(asynchronous=False) ou consumer.commit(message=msg, asynchronous=True)  
  
except KeyboardInterrupt:  
 print("Consumo interrompido pelo usuário.")  
finally:  
 # Fecha o consumidor de forma limpa  
 consumer.close()  
 print("Consumidor fechado.")

**Explicação do Consumidor:**

* **Configuração Essencial:** bootstrap.servers para conectar, group.id para identificar o grupo de consumidores, e auto.offset.reset para definir o comportamento de início de leitura.17
* **Consumer(conf):** Cria uma instância do consumidor.
* **consumer.subscribe([topic]):** Inscreve o consumidor no(s) tópico(s) especificado(s).17
* **Loop de consumer.poll(timeout):** O coração do consumidor. Ele busca mensagens do Kafka.
  + Retorna uma Message se uma mensagem for recebida, ou None se o timeout expirar.
  + Também lida com eventos internos como rebalanceamentos de grupo e callbacks.17
* **Tratamento de Erros:** msg.error() é verificado para lidar com erros ou eventos como \_PARTITION\_EOF (fim da partição).
* **Processamento da Mensagem:** Se msg.error() for None, a mensagem (msg.key(), msg.value()) é processada.
* **consumer.close():** É crucial fechar o consumidor de forma limpa para garantir que os offsets sejam commitados (se o auto-commit estiver habilitado ou para commits finais) e os recursos sejam liberados.17

Para executar estes exemplos:

1. Certifique-se de que o Kafka está rodando (ex: via Docker como na seção anterior).
2. Crie o tópico meu-topico-python se ele não existir (pode ser feito via kafka-topics.sh ou programaticamente).
3. Execute python consumer.py em um terminal.
4. Execute python producer.py em outro terminal.
5. Observe as mensagens sendo produzidas e consumidas.

Estes exemplos básicos demonstram os fundamentos da interação com o Kafka usando Python e a biblioteca confluent-kafka-python, cobrindo a configuração essencial, a produção de mensagens com produce() e o loop de consumo com poll().

## **11. Conclusão**

O Apache Kafka emergiu como uma tecnologia fundamental no cenário de dados moderno, transcendendo sua concepção original como um sistema de mensageria para se tornar uma plataforma robusta e versátil de streaming de eventos. Sua arquitetura distribuída, baseada em logs particionados e replicados, oferece uma combinação única de alta vazão, baixa latência, durabilidade e escalabilidade horizontal, características essenciais para lidar com o volume e a velocidade dos dados gerados atualmente.

A capacidade de reter mensagens por longos períodos e permitir que múltiplos consumidores processem os mesmos dados independentemente abriu portas para uma vasta gama de casos de uso, desde pipelines de dados em tempo real, agregação de logs e métricas, até arquiteturas complexas como event sourcing e microsserviços orientados a eventos. A evolução para o modo KRaft simplificou ainda mais sua operação, eliminando a dependência do ZooKeeper e melhorando a escalabilidade e o desempenho do gerenciamento de metadados.

O ecossistema ao redor do Kafka, com ferramentas como Kafka Connect para integração de dados, Kafka Streams para processamento de fluxos embarcado e Schema Registry para governança de esquemas, enriquece ainda mais suas capacidades, permitindo que as organizações construam soluções de dados ponta a ponta de forma mais eficiente e confiável.

No entanto, o poder e a flexibilidade do Kafka vêm com a necessidade de um entendimento profundo de seus conceitos fundamentais – tópicos, partições, offsets, produtores, consumidores, grupos de consumidores, replicação e segurança. A configuração e o monitoramento adequados, seguindo as melhores práticas para design de tópicos, otimização de desempenho e segurança, são cruciais para o sucesso de qualquer implantação Kafka em produção.

Em comparação com outras tecnologias de mensageria, o Kafka se destaca em cenários que exigem o processamento de grandes volumes de fluxos de eventos com garantias de durabilidade e a capacidade de replay, enquanto sistemas como RabbitMQ podem ser mais adequados para enfileiramento de tarefas tradicionais com roteamento complexo, e AWS SQS para soluções de fila gerenciadas e simples no ecossistema AWS.

A sessão prática demonstra que, com ferramentas como Docker e bibliotecas cliente como confluent-kafka-python, é acessível começar a desenvolver e experimentar com o Kafka, pavimentando o caminho para a construção de aplicações orientadas a eventos cada vez mais sofisticadas.

Em suma, o Apache Kafka não é apenas uma ferramenta, mas uma mudança de paradigma na forma como os dados são tratados, permitindo que as empresas capturem, processem e reajam a eventos em tempo real, impulsionando a inovação e a tomada de decisões baseada em dados. Sua contínua evolução e adoção generalizada solidificam seu papel como um pilar central nas arquiteturas de dados do presente e do futuro.

#### Referências citadas

1. What is Apache Kafka? | IBM, acessado em junho 13, 2025, <https://www.ibm.com/think/topics/apache-kafka>
2. Apache Kafka - Wikipedia, acessado em junho 13, 2025, <https://en.wikipedia.org/wiki/Apache_Kafka>
3. What Is Apache Kafka Used For Everything You Need To Know ..., acessado em junho 13, 2025, <https://www.netdata.cloud/academy/what-is-apache-kafka/>
4. From Producer to Consumer: Understanding Kafka's Message Flow ..., acessado em junho 13, 2025, <https://www.codefro.com/2023/10/03/from-producer-to-consumer-understanding-kafkas-message-flow/>
5. Apache Kafka tutorial: Get started with Kafka in 5 simple steps - NetApp Instaclustr, acessado em junho 13, 2025, <https://www.instaclustr.com/education/apache-kafka/apache-kafka-tutorial-get-started-with-kafka-in-5-simple-steps/>
6. Cluster e componentes do Apache Kafka - IBM, acessado em junho 13, 2025, <https://www.ibm.com/docs/pt-br/SSPFMY_1.3.6/com.ibm.scala.doc/config/iwa_cnf_scldc_apche_con_c.html>
7. Apache Kafka cluster e componentes - IBM, acessado em junho 13, 2025, <https://www.ibm.com/docs/pt-br/oala/1.3.8?topic=components-apache-kafka-cluster>
8. Kafka offset - Redpanda, acessado em junho 13, 2025, <https://www.redpanda.com/guides/kafka-architecture-kafka-offset>
9. Apache Kafka – Wikipédia, a enciclopédia livre, acessado em junho 13, 2025, <https://pt.wikipedia.org/wiki/Apache_Kafka>
10. Kafka Consumer Offsets Guide—Basic Principles, Insights & Enhancements - Confluent, acessado em junho 13, 2025, <https://www.confluent.io/blog/guide-to-consumer-offsets/>
11. How Kafka Consumers Work | Apache Kafka® 101, acessado em junho 13, 2025, <https://developer.confluent.io/courses/apache-kafka/consumers/>
12. Apache Kafka Partition Strategy: Optimizing Data Streaming at Scale, acessado em junho 13, 2025, <https://www.confluent.io/learn/kafka-partition-strategy/>
13. Apache Kafka Partition Key: A Comprehensive Guide - Confluent, acessado em junho 13, 2025, <https://www.confluent.io/learn/kafka-partition-key/>
14. Kafka consumer group - Redpanda, acessado em junho 13, 2025, <https://www.redpanda.com/guides/kafka-architecture-kafka-consumer-group>
15. Differences Between KRaft mode and ZooKeeper mode - Apache Kafka, acessado em junho 13, 2025, <https://kafka.apache.org/40/documentation/zk2kraft.html>
16. Kafka Control Plane: ZooKeeper, KRaft, and Managing Data, acessado em junho 13, 2025, <https://developer.confluent.io/courses/architecture/control-plane/>
17. confluent-kafka-python/examples/sasl\_producer.py at master - GitHub, acessado em junho 13, 2025, <https://github.com/confluentinc/confluent-kafka-python/blob/master/examples/sasl_producer.py>
18. Kafka min.insync.replicas < replication.factor - Codemia, acessado em junho 13, 2025, <https://codemia.io/knowledge-hub/path/kafka_mininsyncreplicas__replicationfactor>
19. Kafka latency-How to reduce it? - Redpanda, acessado em junho 13, 2025, <https://www.redpanda.com/guides/kafka-performance-kafka-latency>
20. Kafka throughput—Trade-offs, solutions and alternatives - Redpanda, acessado em junho 13, 2025, <https://www.redpanda.com/guides/kafka-alternatives-kafka-throughput>
21. How to optimize a Kafka producer for throughput, acessado em junho 13, 2025, <https://developer.confluent.io/confluent-tutorials/optimize-producer-throughput/kafka/>
22. Kafka Performance Tuning: Tips & Best Practices - AutoMQ, acessado em junho 13, 2025, <https://www.automq.com/blog/apache-kafka-performance-tuning-tips-best-practices>
23. APACHE KAFKA: Increasing Replication Factor - Orchestra, acessado em junho 13, 2025, <https://www.getorchestra.io/guides/apache-kafka-increasing-replication-factor>
24. Introducing Horizontal Scaling for DigitalOcean Managed Kafka, acessado em junho 13, 2025, <https://www.digitalocean.com/blog/digitalocean-managed-kafka-scalable-storage-and-additional-nodes>
25. Scaling Your Kafka Cluster A Strategic Approach - Scaler Topics, acessado em junho 13, 2025, <https://www.scaler.com/topics/kafka-tutorial/scaling-kafka-cluster/>
26. What is Kafka Connect? Concepts & Best Practices - AutoMQ, acessado em junho 13, 2025, <https://www.automq.com/blog/kafka-connect-architecture-concepts-best-practices>
27. Quick Start for Confluent Platform, acessado em junho 13, 2025, <https://docs.confluent.io/platform/current/get-started/platform-quickstart.html>
28. axual.com, acessado em junho 13, 2025, <https://axual.com/blog/understanding-kafka-connect#:~:text=Here%20are%20some%20key%20use,Data%20Synchronization>
29. What is Kafka Streams? | Dremio, acessado em junho 13, 2025, <https://www.dremio.com/wiki/kafka-streams/>
30. www.dremio.com, acessado em junho 13, 2025, <https://www.dremio.com/wiki/kafka-streams/#:~:text=Kafka%20Streams%20is%20used%20in,comprehensive%20system%20for%20event%20processing.>
31. www.dremio.com, acessado em junho 13, 2025, <https://www.dremio.com/wiki/confluent-schema-registry/#:~:text=Confluent%20Schema%20Registry%20is%20a,checks%20and%20schema%20evolution%20patterns.>
32. Bitnami package for Confluent Schema Registry - Broadcom Tech Docs, acessado em junho 13, 2025, <https://techdocs.broadcom.com/us/en/vmware-tanzu/application-catalog/tanzu-application-catalog/services/tac-doc/apps-apps-containers-schema-registry-index.html>
33. Confluent Schema Registry | Dremio, acessado em junho 13, 2025, <https://www.dremio.com/wiki/confluent-schema-registry/>
34. A Deep Dive into Apache Kafka's Advanced Capabilities - CelerData, acessado em junho 13, 2025, <https://celerdata.com/glossary/a-deep-dive-into-apache-kafkas-advanced-capabilities>
35. Using Apache Kafka for log aggregation - Redpanda, acessado em junho 13, 2025, <https://www.redpanda.com/guides/kafka-use-cases-log-aggregation>
36. Fluentd vs Kafka - Codemia, acessado em junho 13, 2025, <https://codemia.io/knowledge-hub/path/fluentd_vs_kafka>
37. QuantaStor Log Aggregation with Fluentd – www | grep storage - OSNexus, acessado em junho 13, 2025, <https://blog.osnexus.com/2025/02/11/quantastor-log-aggregation-with-fluentd/>
38. Real-Time Log Aggregation and Analysis for Cybersecurity - Confluent, acessado em junho 13, 2025, <https://www.confluent.io/use-case/log-aggregation-analysis-cybersecurity/>
39. Kafka Metrics - Datadog Docs, acessado em junho 13, 2025, <https://docs.datadoghq.com/opentelemetry/integrations/kafka_metrics/>
40. Monitor Kafka with JMX | Confluent Documentation, acessado em junho 13, 2025, <https://docs.confluent.io/platform/7.2/kafka/monitoring.html>
41. Monitor Kafka with Metrics Reporter in Confluent Platform, acessado em junho 13, 2025, <https://docs.confluent.io/platform/current/monitor/metrics-reporter.html>
42. Generate Metrics for Kafka - Datadog Docs, acessado em junho 13, 2025, <https://docs.datadoghq.com/observability_pipelines/set_up_pipelines/generate_metrics/kafka/>
43. Kafka ACLs Authorization: Usage & Best Practices · AutoMQ/automq ..., acessado em junho 13, 2025, <https://github.com/AutoMQ/automq/wiki/Kafka-ACLs-Authorization:-Usage-&-Best-Practices>
44. Kafka SASL Authentication: Usage & Best Practices - AutoMQ, acessado em junho 13, 2025, <https://www.automq.com/blog/kafka-sasl-authentication-usage-best-practices>
45. Kafka vs RabbitMQ: Key Differences & When to Use Each | DataCamp, acessado em junho 13, 2025, <https://www.datacamp.com/blog/kafka-vs-rabbitmq>
46. RabbitMQ vs. Apache Kafka | Confluent, acessado em junho 13, 2025, <https://www.confluent.io/learn/rabbitmq-vs-apache-kafka/>
47. Kafka vs SQS | Svix Resources, acessado em junho 13, 2025, <https://www.svix.com/resources/faq/kafka-vs-sqs/>
48. Kafka vs. SQS: A Deep Dive into Messaging and Streaming Platforms - AutoMQ, acessado em junho 13, 2025, <https://www.automq.com/blog/kafka-vs-sqs-messaging-streaming-platforms-comparison>
49. How to run Kafka locally with Docker - Confluent Developer, acessado em junho 13, 2025, <https://developer.confluent.io/confluent-tutorials/kafka-on-docker/>
50. docker-compose.yml - confluentinc/confluent-kafka-python - GitHub, acessado em junho 13, 2025, <https://github.com/confluentinc/confluent-kafka-python/blob/master/examples/fips/docker/docker-compose.yml>