Diagrama de Arquitetura – Fluxo End-to-End

```
flowchart LR
    %% =====================
    %% PRODUCERS
    %% =====================
    subgraph PROD["Fontes / Producers (.NET em EC2)"]
        P1["Producer .NET<br/>- Extrai código<br/>- Define key =
código<br/>- JSON ~200 bytes"]
    end

    %% =====================
    %% KAFKA / MSK
    %% =====================
    subgraph MSK["Amazon MSK (Kafka)"]
        subgraph T1["Tópico mt-c400"]
            P400_0["Partição 0<br/>(código 300)"]
            P400_1["Partição 1<br/>(código 301)"]
            P400_2["Partição 2<br/>(código 302)"]
            P400_3["Partição 3<br/>(código 303)"]
            P400_N["Partições extras<br/>(códigos mais quentes)"]
        end

        subgraph T2["Tópico mt-c300"]
            P300_0["Partição 0"]
            P300_1["Partição 1"]
        end
    end

    %% =====================
    %% CONSUMERS
    %% =====================
    subgraph CONS["Consumers (.NET em EC2)<br/>Consumer Group"]
        C1["Consumer EC2 #1<br/>Consome 1+ partições"]
        C2["Consumer EC2 #2<br/>Consome 1+ partições"]
        C3["Consumer EC2 #N<br/>Escala horizontal"]
    end

    %% =====================
    %% LAMBDA
    %% =====================
    subgraph AWS["AWS Lambda"]
        L1["Lambda Dispatcher<br/>- Recebe lote<br/>- Roteia para
serviços"]
    end

    %% =====================
    %% FLUXOS
    %% =====================
    P1 -->|Produce com key=código| MSK
```

```
    P400_0 --> C1
    P400_1 --> C2
    P400_2 --> C3
    P400_3 --> C1
    P400_N --> C2

    P300_0 --> C3
    P300_1 --> C1

    C1 -->|Batch JSON| L1
    C2 -->|Batch JSON| L1
    C3 -->|Batch JSON| L1
```

1️⃣ Producer (.NET em EC2)

- O Producer **não cria partições dinamicamente**.
- Ele apenas:
  - Lê a mensagem
  - Extrai o `codigo`
  - Define `key = codigo`
- O Kafka garante que **a mesma key sempre vai para a mesma partição**.

👉 **Decisão importante**:
A lógica de distribuição está no **Producer**, não no MSK.

2️⃣ Tópicos Kafka (Domínio de Negócio)

- Cada tópico representa um **domínio funcional** (`mt-c400`, `mt-c300`).
- Não há um tópico por código → evita explosão de tópicos.
- O tópico mais pesado (`mt-c400`) possui **mais partições**.

👉 **Argumento forte**:

> "Tópicos representam domínios, partições representam escala."

3️⃣ Partições (Escala e Paralelismo)

- Cada partição é:
  - Unidade de paralelismo
  - Unidade de ordenação
- Códigos mais frequentes podem:

- Ter partições dedicadas
- Ou múltiplas partições no futuro (sub-key)

👉 **Trade-off consciente**:
Possibilidade de hot partition, mitigável com monitoramento.

4️⃣ Consumers em EC2 (.NET)

- Consumers fazem parte de **um único consumer group**.
- Kafka distribui automaticamente as partições.
- Escalar = subir mais EC2s.
- Cada consumer:
  - Consome mensagens
  - Agrupa em lote
  - Confirma offset após sucesso

👉 **Benefício**:
Escala horizontal sem mudança de código.

5️⃣ Lambda (Fan-out e Distribuição)

- Recebe mensagens **em lote**.
- Reduz custo de invocação.
- Faz roteamento para serviços downstream.
- Mantém consumer simples e rápido.

👉 **Separação clara de responsabilidades**:

- Kafka = throughput
- EC2 = processamento contínuo
- Lambda = distribuição e elasticidade

Mensagem-chave para convencer os pares

> “Essa arquitetura desloca a complexidade para onde temos controle (Producer e Consumer) e mantém o MSK simples, estável e escalável. Ela evita explosão de tópicos, permite crescimento previsível e é compatível com volumes industriais de mensagens.”

Ótimo ponto. Como você está desenhando **arquitetura de alto throughput com Kafka + MSK + .NET + Lambda**, existem **conceitos fundamentais (teóricos e práticos)** que valem muito a pena você dominar e até **defender tecnicamente** em reuniões, ADRs ou banca acadêmica. Vou organizar em **camadas conceituais**, sempre conectando com o seu cenário real (13k msg/s, Pareto 80/20, múltiplos tópicos e partições).

---

1. Conceitos fondamentais de Kafka que você DEVE abordar

1.1 Tópico ≠ Fila (log distribuído)

Conceito

- Kafka não é uma fila tradicional, é um **log distribuído e particionado**.
- Mensagens **não são removidas ao serem consumidas**.
- Cada consumidor mantém seu **offset**.

Por que isso importa no seu caso

- Você pode ter **vários consumidores lendo o mesmo tópico** (ex: analytics, auditoria, distribuição).
- O Lambda não "consome" a mensagem, ele **processa uma posição do log**.
- Reprocessamento é possível (reset de offset).

Frase-chave para defender

> "Kafka desacopla produtores e consumidores usando um log imutável, o que nos dá replay, escalabilidade e tolerância a falhas."

1.2 Partições = unidade real de paralelismo

Conceito

- A **partição é a menor unidade de paralelismo** no Kafka.
- Uma partição → **1 consumidor por consumer group**.
- Não existe paralelismo dentro de uma partição.

No seu cenário

- Se você quer processar **13.000 msg/s**, precisa dividir isso em **fatias paralelas**.
- Exemplo:
  - 13.000 / 13 partições ≈ 1.000 msg/s por partição
  - 13 consumidores em paralelo

Decisão arquitetural

- `mt-c400` → muitas partições
- `mt-c300` → poucas partições

Frase forte

> "Escalar Kafka é escalar partições, não CPU."

1.3 Consumer Group como mecanismo de balanceamento automático

Conceito

- Um **consumer group** garante:
    - Balanceamento automático de partições
    - Failover transparente
    - Escalabilidade horizontal

No seu caso

- Você sobe mais instâncias do serviço C#
- Kafka redistribui partições
- Nenhuma configuração manual

Relação direta com Lambda

- Event Source Mapping do Lambda cria **um consumer group gerenciado pela AWS**

Frase

> "Consumer groups nos dão elasticidade sem coordenação manual."

---

2. Conceitos de particionamento (onde está o ouro)

2.1 Hash vs Key-based partitioning

Hash / Round-robin

- Máximo throughput
- Carga bem distribuída
- Ordem global NÃO garantida

Key-based (ex: código 300, 301…)

- Ordem garantida por chave
- Possível hotspot

No seu cenário

- 80% das mensagens vêm de poucos códigos
- Se particionar só pelo código → risco de gargalo

Decisão madura

- Usar **composite key**:

```
key = codigo + hash(deviceId)
```

Frase técnica

> “Mantemos ordem lógica sem sacrificar paralelismo físico.”

2.2 Hot partitions e Lei de Pareto (80/20)

Conceito

- 20% das chaves geram 80% do tráfego
- Kafka não resolve hotspot sozinho

Soluções arquiteturais

1. Mais partições no tópico quente
2. Tópico dedicado para heavy hitters
3. Sub-particionamento lógico
4. Sharding por tempo ou região

Exemplo

- `mt-c400-heavy`
- `mt-c400-light`

Frase

> “Identificamos hot keys e tratamos como first-class citizens da arquitetura.”

---

3. Conceitos de throughput e performance

3.1 Batching (o conceito mais subestimado)

Conceito

- Kafka é otimizado para **lotes**, não mensagens individuais
- Produzir e consumir em batch aumenta throughput em ordens de magnitude

No consumidor C#

- `max.poll.records`
- Processar arrays
- Enviar batches ao Lambda

Frase

> “Nós otimizamos chamadas, não mensagens.”

3.2 Backpressure (fundamental para estabilidade)

Conceito

- O sistema **precisa saber desacelerar**

- Sem isso → colapso em cascata

No seu pipeline

```
Kafka → C# → Lambda → Serviços
```

Estratégias

- Pausar consumo (`Pause/Resume`)
- Controlar batch size
- Limitar concorrência do Lambda
- DLQ para falhas

Frase

> "Preferimos atrasar processamento a perder dados."

---

4. Conceitos de semântica de entrega

4.1 At-least-once vs At-most-once vs Exactly-once

Kafka por padrão

- At-least-once

O que isso significa

- Mensagens podem ser reprocessadas
- Seu downstream precisa ser **idempotente**

No Lambda

- Use:
    - Deduplicação
    - Chave de idempotência
    - DynamoDB conditional writes

Frase

> "Garantimos consistência por design, não por sorte."

---

5. Kafka + Lambda: choque de modelos

5.1 Pull (Kafka) vs Push (Lambda)

Kafka

- Pull-based

- Controle de ritmo no consumidor

Lambda

- Push-based
- Escala agressivamente

Conflito

- Lambda pode escalar mais rápido que downstream

Solução

- Batch controlado
- Concurrency limit
- Step Functions ou SQS buffer se necessário

Frase

> “Integramos dois modelos opostos controlando o ponto de acoplamento.”

---

6. Conceitos de observabilidade (nível sênior)

6.1 Lag como SLO

Conceito

- Lag = backlog real
- Métrica mais importante do Kafka

Você deve monitorar

- Lag por partição
- Throughput por consumer
- Tempo Kafka → Lambda

Frase

> “Lag é nosso termômetro operacional.”

6.2 Tracing distribuído

Conceito

- Uma mensagem cruza:
    - Kafka
    - C#
    - Lambda
    - Serviços

Solução

- Correlation ID
- OpenTelemetry
- Logs estruturados

Frase

> “Observabilidade não é opcional em sistemas distribuídos.”

---

7. Conceitos de arquitetura (para fechar com chave de ouro)

7.1 Desacoplamento por eventos

Você está construindo

- Uma **arquitetura orientada a eventos**
- Não RPC, não request/response

Benefícios

- Escalabilidade
- Evolução independente
- Resiliência

7.2 Design para falha

Assumir

- Broker cai
- Consumer cai
- Lambda falha

Kafka permite

- Replay
- Retenção
- Reprocessamento controlado

Frase final forte

> “Não evitamos falhas, projetamos para sobreviver a elas.”

Se quiser ir além (nível doutorado / staff engineer)

Posso te ajudar a:

- Criar **ADR formal** dessas decisões
- Calcular **número ótimo de partições**

- Simular throughput em Python
- Criar um **diagrama C4**
- Escrever isso como **capítulo de tese ou whitepaper**

Perfeito. Abaixo está um **diagrama em Mermaid** focado **exatamente nos pontos de tensão da discussão arquitetural**:
isolamento vs compartilhamento, blast radius, prioridade, chave de partição e evolução para arquitetura híbrida.

A ideia não é só mostrar o "fluxo feliz", mas **visualizar os riscos e as decisões**.

1️⃣ Visão Geral – Arquitetura Atual (Domínio + Partições)

```
flowchart LR
    subgraph PROD["Producers (.NET em EC2)"]
        P["Producer<br/>- Extrai código<br/>- Define key (ex: código)
<br/>- JSON 200 bytes"]
    end

    subgraph MSK["Amazon MSK"]
        subgraph C400["Tópico mt-c400 (mesmo domínio)"]
            C400_P0["Partição 0<br/>Código 300"]
            C400_P1["Partição 1<br/>Código 301"]
            C400_P2["Partição 2<br/>Código 302"]
            C400_PN["Partições extras<br/>Códigos quentes"]
        end
    end

    subgraph CONS["Consumer Group (.NET EC2)"]
        C1["Consumer #1"]
        C2["Consumer #2"]
        C3["Consumer #N"]
    end

    subgraph LAMBDA["AWS Lambda"]
        L["Lambda Dispatcher<br/>- Batch<br/>- Roteia por código"]
    end

    P -->|key = código| C400
    C400_P0 --> C1
    C400_P1 --> C2
    C400_P2 --> C3
    C400_PN --> C1

    C1 -->|batch| L
    C2 -->|batch| L
    C3 -->|batch| L
```

🧠 O que este diagrama explica

- **Partições são a unidade de escala**, não tópicos
- Ordem é garantida **por código**, não global
- Consumer é genérico → reduz filtering cost
- Lambda isola lógica e fan-out

2️⃣ Onde mora o risco – Blast Radius e Hot Partition

```
flowchart LR
    subgraph MSK["Tópico mt-c400"]
        P0["Partição X<br/>Código 300<br/>(ALTA PRIORIDADE)"]
        P1["Partição Y<br/>Código 301<br/>(Volume Alto)"]
    end

    subgraph CONS["Consumer EC2"]
        C["Consumer"]
    end

    P1 -->|Mensagem malformada| C
    C -.->|Erro não tratado| P1
    P1 -.->|Lag aumenta| P0

    style P1 fill:#ffdddd
    style P0 fill:#fff4cc
```

⚠️ O que este diagrama evidencia

- Um erro em dados **não críticos** pode:
  - Aumentar lag
  - Afetar processamento crítico
- Esse risco **só existe** se:
  - Consumer tiver lógica pesada
  - Erro travar partição

👉 **Mitigação**:

- Consumer simples
- DLQ
- Try/catch por mensagem

3️⃣ Comparação visual – Muitos Tópicos vs Poucos Tópicos

```
flowchart TB
    subgraph A["Estratégia A<br/>Muitos Tópicos"]
        A300["topic-code-300"]
        A301["topic-code-301"]
        A302["topic-code-302"]
    end

    subgraph B["Estratégia B<br/>Domínio + Partições"]
        B400["mt-c400"]
        B400_P["Partições<br/>300 | 301 | 302"]
    end

    style A fill:#ffe6e6
    style B fill:#e6f2ff
```

📊 Leitura arquitetural

- Estratégia A:
  - Alto isolamento
  - Alto custo operacional
  - Topic sprawl
- Estratégia B:
  - Bom isolamento lógico
  - Escala eficiente
  - Governança centralizada

4️⃣ Evolução Natural – Arquitetura Híbrida (Prioridade)

```
flowchart LR
    subgraph PROD["Producer (.NET)"]
        P["Producer<br/>Decide tópico<br/>com base na prioridade"]
    end

    subgraph MSK["Amazon MSK"]
        HP["high_priority_events<br/>(Código 300)"]
        STD["telemetry_standard<br/>(301, 302)"]
    end

    subgraph CONS["Consumers EC2"]
        C_HP["Consumers Alta Prioridade"]
        C_STD["Consumers Standard"]
    end

    subgraph LAMBDA["Lambda"]
        L["Dispatcher"]
```

```
    end

    P -->|300| HP
    P -->|301/302| STD

    HP --> C_HP
    STD --> C_STD

    C_HP --> L
    C_STD --> L
```

🧠 Mensagem-chave

- Evolução **não quebra o desenho atual**
- Producer já controla:
  - Key
  - Tópico
- Isolamento cresce conforme criticidade

5️⃣ Decisão sobre chave – `truck_id` vs `codigo`

```
flowchart LR
    subgraph KEY1["key = truck_id"]
        K1["300 + 301<br/>mesma partição"]
    end

    subgraph KEY2["key = código"]
        K2["300 → partição A"]
        K3["301 → partição B"]
    end

    style KEY1 fill:#ffe6e6
    style KEY2 fill:#e6ffe6
```

📌 Interpretação

- `truck_id`:
  - Ordem forte
  - Alto acoplamento
- `codigo`:
  - Melhor isolamento
  - Ordem suficiente

- Menor blast radius

6️⃣ Diagrama mental para defender a arquitetura

```
flowchart TB
    A["Domínio bem definido"] --> B["Poucos tópicos"]
    B --> C["Partições para escala"]
    C --> D["Producer controla key"]
    D --> E["Consumer simples"]
    E --> F["Lambda isola fan-out"]
    F --> G["Evolução para isolamento por prioridade"]
```

🎯 Como usar esse material na discussão

Você pode literalmente apontar para os diagramas e dizer:

- “Aqui está onde assumimos risco conscientemente”
- “Aqui está como mitigamos”
- “Aqui está como evoluímos sem reescrever tudo”

Abordagem do producer

1. Objetivo da POC
2. Responsabilidades do Producer
3. Arquitetura lógica do Producer
4. Estratégia para tópicos e partições
5. Escala: 1 instância vs múltiplas instâncias
6. Controle, observabilidade e governança
7. Fluxograma (Mermaid)
8. Etapas de implementação
9. Pontos perigosos e riscos reais

🎯 1. Objetivo da POC (Producer)

O que a POC PRECISA provar

A POC do **Producer EC2 .NET** deve responder objetivamente:

- Consigo **publicar corretamente** em múltiplos tópicos?
- Consigo **controlar a distribuição de mensagens por partição**?
- Consigo **simular volume realista**?

- Consigo **escalar horizontalmente sem quebrar nada**?
- Consigo **medir impacto em latência e custo**?

⚠ **Importante**:
A POC **não** precisa:

- Ser altamente resiliente
- Ter autoscaling perfeito
- Ter HA completo

Ela precisa **validar decisões arquiteturais**, não ser produção.

📦 2. Responsabilidades do Producer

O Producer **não é burro**, mas também **não deve ser inteligente demais**.

Responsabilidades corretas

✔ Escolher o tópico
✔ Definir a key de partição
✔ Garantir confiabilidade (`acks=all`)
✔ Controlar taxa (throttling)
✔ Emitir métricas

Responsabilidades que NÃO são dele

❌ Saber quantas partições existem
❌ Fazer load balancing manual
❌ Garantir ordem global
❌ Saber quem vai consumir

👉 **Kafka já faz isso melhor.**

🧠 3. Arquitetura lógica do Producer

Pense no Producer como **pipeline**, não como um "serviço REST".

```
Input Generator
      ↓
Message Builder
      ↓
Topic Resolver
      ↓
Partition Key Resolver
      ↓
```

```
Kafka Producer
      ↓
Metrics & Logs
```

🧩 4. Estratégia para tópicos e partições

🔑 Regra de ouro

> **Producer escolhe a KEY, Kafka escolhe a PARTIÇÃO**

Exemplo

```
Topic: mt-c400
Key: hash(codigo + truck_id)
```

Kafka:

- Aplica hash(key)
- Mod N (partições)
- Distribui automaticamente

⚠ Você **não** deve usar `partition=X` manualmente na POC.

🔍 Estratégia de chave (decisão crítica)

Opções

Estratégia	Prós	Contras
`truck_id`	Ordem por caminhão	Hot partitions
`codigo`	Isolamento lógico	Perda de ordem
`hash(codigo+truck)`	Equilíbrio	Ordem parcial

👉 **Para POC:**
Use `hash(codigo + truck_id)`
É a mais defensável.

⚖️ 5. Escala: quantas instâncias de Producer?

❗ Verdade importante

> **Kafka Producer escala melhor por THREAD do que por INSTÂNCIA**

Estratégia correta para POC

Fase 1 – 1 EC2

- 1 processo
- 1 Producer
- 5–10 threads
- Controle de TPS

Fase 2 – 2 EC2

- Mesma config
- Mesmos tópicos
- Mesmas chaves

👉 Kafka garante que:

- Não há duplicação
- Não há conflito
- Não há desordem por key

⚠ Erro comum

> "Vou criar um Producer por partição"

❌ ERRADO
Isso quebra:

- Escala
- Rebalance
- Custo

🎛 6. Controle e governança

Controle de taxa (obrigatório)

```
- Mensagens/segundo
- Bytes/segundo
```

Implemente:

- Token bucket simples
- Sleep controlado

- Config via ENV VAR

Observabilidade mínima

Métrica	Por quê
msg/s	Throughput
ack latency	Saúde MSK
error rate	Confiabilidade
retries	Saturação

🔁 7. Fluxograma do Producer (Mermaid)

```
flowchart TD
    A[Start Producer] --> B[Load Config]
    B --> C[Initialize Kafka Producer]
    C --> D[Start Worker Threads]

    D --> E[Generate Payload]
    E --> F[Resolve Topic]
    F --> G[Resolve Key]
    G --> H[Send to Kafka]

    H --> I{Ack OK?}
    I -->|Yes| J[Emit Metrics]
    I -->|No| K[Retry / Log Error]

    J --> E
```

🛠️ 8. Etapas práticas da POC

Etapa 1 – Infra mínima

- EC2 t3.medium
- Security Group liberando MSK
- Certificados funcionando

Etapa 2 – Código Producer .NET

Componentes:

- `ProducerFactory`
- `MessageGenerator`
- `TopicRouter`
- `KeyResolver`
- `MetricsEmitter`

Etapa 3 – Simulação de carga

- Ramp-up gradual
- 1 → 5 → 10 threads
- Observa latência

Etapa 4 – Scale-out

- Sobe segunda EC2
- Mesmo código
- Observa:
  - Distribuição
  - Throughput
  - Impacto no MSK

☠️ 9. Pontos perigosos e riscos reais

🔥 Risco 1 – Hot partitions

- Chave mal escolhida
- Solução: hash composta

🔥 Risco 2 – Buffer interno do Producer

- `queue.buffering.max.messages`
- Pode causar OOM
- Monitorar

🔥 Risco 3 – Backpressure invisível

- MSK lento
- Producer continua aceitando
- Latência explode

🔥 Risco 4 – Overengineering

- Producer “inteligente demais”
- Lógica duplicada
- Difícil de manter

🔥 Risco 5 – Falta de métricas

> “Sem métricas, você não tem arquitetura, só esperança.”

🧠 Conclusão arquitetural

> O Producer **não controla o Kafka**.
> Ele **declara intenção** (key, tópico, confiabilidade).
> O Kafka **executa a estratégia de escala**.

Essa POC vai te permitir:

- Defender a arquitetura com números
- Antecipar gargalos
- Calibrar decisões antes da produção

➡️ SOON Próximo passo (se quiser)

Posso:

- Criar **pseudocódigo .NET detalhado do Producer**
- Criar **checklist de POC**
- Criar **template de relatório de resultados**
- Avançar para **EC2 Consumer**

Se quiser, diga:
👉 *“Vamos detalhar o código do Producer”*

Perfeito. Vou te entregar **dois diagramas complementares**, no nível que normalmente aparece em **documentação de arquitetura séria**:

1. **Diagrama arquitetural do Producer (visão estrutural)**
2. **Fluxograma detalhado de execução do Producer (visão comportamental)**

Ambos pensados para:

- Sustentar discussão com arquitetos, SRE e FinOps
- Ajudar alguém a **construir o serviço do zero**
- Deixar explícito **onde estão os riscos e controles**

🧱 1️⃣ Diagrama Arquitetural do Serviço Producer (.NET em EC2)

👉 **Pergunta que esse diagrama responde**

> “Que componentes existem dentro do Producer e como eles se relacionam?”

```
flowchart LR
    subgraph EC2["EC2 Producer (.NET)"]
        CFG["Config Loader<br/>(ENV / AppSettings)"]
        GEN["Message Generator<br/>(Simula fontes)"]
        ROUTER["Topic Router<br/>(Regra por código)"]
        KEY["Key Resolver<br/>(hash(codigo+truck))"]
        KPROD["Kafka Producer<br/>(librdkafka)"]
        MET["Metrics & Logs<br/>(CloudWatch / OTEL)"]
        RATE["Rate Limiter<br/>(TPS / Bytes)"]
    end

    subgraph MSK["Amazon MSK"]
        T400["Topic mt-c400<br/>(N Partições)"]
        T300["Topic mt-c300<br/>(M Partições)"]
    end

    CFG --> GEN
    GEN --> ROUTER
    ROUTER --> KEY
    KEY --> RATE
    RATE --> KPROD
    KPROD --> T400
    KPROD --> T300
    KPROD --> MET
    RATE --> MET
```

📌 Como explicar esse diagrama em reunião

- **Config Loader**
  → Nada hardcoded. TPS, tópicos e comportamento são configuráveis.
- **Message Generator**
  → Simula múltiplas fontes e Pareto (80/20).
- **Topic Router**
  → Decide *qual domínio* (mt-c400, mt-c300).
- **Key Resolver**
  → Define *como Kafka irá distribuir*.

- **Rate Limiter**
  → Protege o MSK **e o próprio Producer**.
- **Kafka Producer**
  → Único ponto de comunicação com o cluster.
- **Metrics**
  → Sem isso, a POC não vale nada.

🔁 2️⃣ Fluxograma Detalhado do Producer (Execução passo a passo)

👉 **Pergunta que esse fluxograma responde**

> “O que acontece exatamente desde o start até o envio contínuo?”

```
flowchart TD
    A[Start EC2 Producer] --> B[Load Configurations]
    B --> C[Validate Configs]
    C -->|Invalid| CERR[Fail Fast & Exit]
    C -->|Valid| D[Init Kafka Producer]

    D --> E[Init Metrics & Logs]
    E --> F[Start Worker Threads]

    F --> G[Generate Message]
    G --> H[Extract Business Code]
    H --> I[Resolve Topic]
    I --> J[Resolve Partition Key]

    J --> K[Rate Limit Check]
    K -->|Exceeded| KWAIT[Wait / Sleep]
    KWAIT --> K
    K -->|Allowed| L[Produce Message]

    L --> M{Ack Received?}
    M -->|Yes| N[Update Metrics]
    M -->|No| O[Retry Policy]

    O -->|Retryable| L
    O -->|Fatal| P[Log Error & Continue]

    N --> G
```

📌 Pontos críticos destacados no fluxograma

🔴 Fail Fast

- Se config errada → **não sobe**
- Evita producer “zumbi”

🔴 Rate Limiter antes do Kafka

- Kafka **não protege você**
- Sem isso:
  - Memória cresce
  - Latência explode
  - EC2 morre

🔴 Retry consciente

- Nem todo erro é retryável
- Erro de schema ≠ erro de rede

⚠ Pontos perigosos explícitos no diagrama

Ponto	Risco	Mitigação
Key mal escolhida	Hot partition	Hash composta
Sem rate limit	OOM	Throttling
Retries infinitos	Loop infinito	Retry com limite
Config fixa	Impossível escalar	ENV vars
Métricas pobres	Arquitetura “cega”	CloudWatch / OTEL

🧠 Como um arquiteto sênior justificaria esse desenho

> “Esse Producer não sabe nada sobre o MSK além do necessário. Ele declara intenção (tópico, chave, confiabilidade) e deixa o Kafka fazer o trabalho pesado. Isso reduz acoplamento, facilita escala e torna o comportamento previsível sob carga.”

📋 Checklist rápido (para POC)

✔ Código configurável
✔ Simulação Pareto
✔ Métricas claras
✔ Rate limit ativo
✔ Chave bem definida
✔ Fácil de escalar horizontalmente

Perfeito. Abaixo está um **PLAYBOOK DE AUMENTO DE PARTIÇÕES KAFKA (MSK)**, escrito no nível que **SRE, arquiteto e time .NET** esperam ver.
Ele cobre **quando fazer**, **impactos reais**, **o que esperar**, **como mitigar**, e **como o código .NET deve se comportar antes e depois**.

Este material é **diretamente reutilizável** na sua documentação oficial.

📘 PLAYBOOK – Aumento de Partições em Kafka (MSK)

🎯 Objetivo

Permitir **aumento de throughput e paralelismo** sem downtime, mantendo:

- previsibilidade operacional
- impacto controlado
- zero mudança no código do Producer/Consumer (.NET)

1️⃣ Quando aumentar partições (critérios objetivos)

Você **NÃO aumenta partições por intuição**.
Você aumenta quando **métricas indicam gargalo estrutural**.

Indicadores claros

Métrica	Sintoma
Consumer Lag cresce continuamente	Capacidade insuficiente
Apenas algumas partições saturadas	Hot partition
CPU de consumers < 50%	Falta paralelismo
Throughput estagnado	Limite físico
Escalar consumers não ajuda	Falta partições

👉 **Regra prática**

> Número de partições ≥ número máximo esperado de consumers ativos

---

2️⃣ O que acontece tecnicamente ao aumentar partições

Estado ANTES

```
Topic mt-c400
Partições: 8
partition = hash(key) % 8
```

Estado DEPOIS

```
Topic mt-c400
Partições: 16
partition = hash(key) % 16
```

Efeitos imediatos

✔ Mensagens antigas permanecem onde estão
✔ Mensagens novas passam a usar novas partições
⚠ A mesma key pode ir para outra partição
⚠ Ordem histórica entre mensagens antigas e novas é quebrada

👉 **Nada é perdido. Nada é duplicado.**

3️⃣ Impactos esperados (e normais)

3.1 Rebalance do Consumer Group

O que acontece

- Todos os consumers pausam
- Kafka redistribui partições
- Consumo retoma

O que esperar

- Pausa de segundos (ou minutos se exagerou nas partições)
- Lag temporário

Mitigação

- Aumentar partições fora do horário de pico
- Garantir que consumers sejam idempotentes

3.2 Redistribuição de carga

Antes

```
Partição 2 → 80% do tráfego
```

Depois

```
Partição 2 → 40%
Partição 10 → 40%
```

👉 **Esse é o ganho esperado.**

3.3 Mudança invisível para o Producer

✔ Producer não sabe
✔ Producer não muda
✔ Producer continua saudável

SE você usou:

- key-based partitioning
- sem partição fixa no código

4️⃣ Riscos reais e como resolver

🔥 RISCO 1 – Explosão de rebalance

Causa

- Muitas partições
- Muitos consumers
- Deploy frequente

Mitigação

- Evitar “partições demais”
- Preferir scale-up antes de scale-out
- Evitar restart em massa

🔥 RISCO 2 – Quebra de expectativa de ordem

Causa

- Key vai para outra partição

Mitigação

- Documentar claramente:

  > “Ordem é garantida apenas após o resize”

- Usar timestamp de evento

🔥 RISCO 3 – Hot partition continua

Causa

- Chave mal escolhida

Mitigação

```
ANTES: key = codigo
DEPOIS: key = hash(codigo + truck_id)
```

5️⃣ Como se preparar ANTES do aumento (obrigatório)

✔ 1. Código .NET deve ser agnóstico a partições

❌ ERRADO

```
new TopicPartition("mt-c400", new Partition(3))
```

✔ CORRETO

```
producer.Produce(
    "mt-c400",
    new Message<string, string>
    {
        Key = $"{codigo}:{truckId}",
        Value = payload
    }
);
```

✔ 2. Consumers devem ser idempotentes

Após rebalance:

- Mensagem pode ser reprocessada
- Offset pode ser replayado

👉 Lambda downstream deve suportar isso.

✔ 3. Métricas obrigatórias antes do resize

- Consumer lag por partição
- Rebalance time
- Throughput por key
- Ack latency do Producer

6️⃣ Passo a passo operacional (Playbook)

🔷 PASSO 1 – Validar necessidade

- Confirmar gargalo
- Confirmar que mais consumers não resolvem

🔷 PASSO 2 – Escolher novo número

Regra prática:

```
Novo N = 2x partições atuais
```

Evite:

- Incrementos pequenos
- Ajustes frequentes

🔷 PASSO 3 – Executar resize

```
kafka-topics.sh \
  --bootstrap-server <broker> \
  --alter \
  --topic mt-c400 \
  --partitions 16
```

✔ Sem downtime
✔ Sem restart

🔷 PASSO 4 – Monitorar rebalance

Acompanhar:

- consumer-group-state
- lag total
- throughput

🔷 PASSO 5 – Validar estabilização

Critério de sucesso:

- Lag volta ao normal
- CPU de consumers sobe
- Throughput aumenta

7️⃣ O que NÃO fazer (anti-patterns)

❌ Fixar partições no Producer
❌ Criar lógica “partição dedicada” no código
❌ Aumentar partições toda semana
❌ Ignorar rebalance
❌ Achar que aumento é “sem impacto”

8️⃣ Pergunta dura de SRE (simulação)

❓ “Se dobrarmos as partições agora, o que pode dar errado?”

Resposta correta

> “Teremos um rebalance temporário e possível quebra de ordem histórica, mas nenhum impacto em integridade, perda de dados ou necessidade de mudança de código.”

9️⃣ Pergunta dura de arquiteto (simulação)

❓ “Por que não criar outro tópico em vez de aumentar partições?”

Resposta madura

> “Porque o domínio não mudou. Só a capacidade. Partições escalam processamento, tópicos escalam contratos.”

🔚 Conclusão para documentação

> “Aumentar partições é uma operação planejada, previsível e segura quando o Producer é key-based, os Consumers são idempotentes e o impacto é monitorado. O risco não está no Kafka, está em arquiteturas que acoplam código à topologia.”

Perfeito. Vou estruturar **um playbook completo para o CONSUMER (.NET em EC2)**, já alinhado com tudo o que discutimos: MSK gerenciado, múltiplos tópicos, múltiplas partições, possibilidade de aumento de partições no futuro, batching e integração com Lambda.

Vou escrever isso no **nível de documentação técnica + operacional**, do jeito que um **SRE, arquiteto ou tech lead espera ler**.

📘 Playbook do Consumer Kafka (.NET + EC2 + MSK)

---

1. Papel do Consumer na Arquitetura

O **consumer** é o componente mais crítico da arquitetura porque ele:

- Converte **throughput Kafka → processamento real**
- Define **latência**, **ordenação**, **paralelismo**
- É o ponto onde **backpressure aparece**
- É onde falhas viram **reprocessamento, duplicação ou perda**

Na sua arquitetura, o consumer:

- Consome tópicos como `mt-c400`
- Lê **N partições em paralelo**
- Agrega mensagens em **lotes**
- Dispara **Lambda** quando:
  - Batch size atingido **OU**
  - Timeout atingido

---

2. Princípios de Design (não negociáveis)

2.1 Consumer Group é obrigatório

> **Escalabilidade em Kafka = consumer group**

- Cada instância EC2 do consumer:
  - Usa o **mesmo** `group.id`
- Kafka garante:
  - **1 partição → 1 consumer ativo**
  - Rebalance automático

👉 Se você tem **12 partições**, o máximo de paralelismo real é **12 consumers ativos**.

2.2 Consumer NÃO decide partição

> Diferente do producer, o consumer **não escolhe partições**

Ele:

- Recebe partições atribuídas pelo **Kafka Group Coordinator**
- Deve ser **agnóstico** à quantidade de partições

Isso é essencial para:

- Suportar **aumento futuro de partições**
- Evitar reescrita do código

---

3. Estratégia de Escala do Consumer

3.1 Relação Partições x Instâncias

Partições	EC2 Consumers	Resultado
6	3	Cada EC2 consome 2 partições
6	6	1 partição por EC2
6	10	4 EC2 ficam ociosos
12	6	Cada EC2 consome 2 partições

👉 **Nunca escale EC2 sem considerar partições**
👉 Escalar além do número de partições **não aumenta throughput**

3.2 Modelo recomendado

- **Auto Scaling Group (ASG)**
- Métricas:
  - Lag por consumer group
  - CPU / memória
- Escala:
  - Scale out **até o limite de partições**
  - Depois disso → só aumenta custo

---

4. Fluxo Lógico do Consumer

4.1 Fluxo macro

1. EC2 sobe
2. Inicializa Kafka Consumer
3. Entra no Consumer Group
4. Kafka atribui partições
5. Loop de consumo:
    - Poll mensagens
    - Bufferiza
    - Dispara Lambda
    - Commita offsets

4.2 Fluxograma (mental)

```
START
 ↓
Connect to MSK (mTLS)
 ↓
Join Consumer Group
 ↓
Get Assigned Partitions
 ↓
WHILE running:
    Poll messages
    Add to batch
    IF batch size OR timeout:
        Call Lambda
        Commit offsets
```

---

5. Estratégia de Batching (fundamental)

5.1 Por que batching?

Sem batching:

- Milhares de invocações Lambda
- Custo alto
- Latência variável

Com batching:

- Controle de custo
- Throughput previsível
- Melhor uso de rede

5.2 Tipos de batching

Você **DEVE** usar dois gatilhos:

1. **Batch Size**
    - Ex: 500 mensagens
2. **Batch Timeout**
    - Ex: 1 segundo

> Nunca use apenas tamanho — isso causa latência infinita em períodos de baixo tráfego

---

6. Commit de Offset (ponto crítico)

6.1 Estratégia correta

- `EnableAutoCommit = false`
- Commit **após Lambda responder OK**

```
consumer.Commit();
```

6.2 Riscos

Situação	Impacto
Commit antes do Lambda	Perda de dados
Commit depois do Lambda	Possível duplicação
Lambda falha	Reprocessamento

👉 **Kafka trabalha com "at-least-once"**
👉 Duplicação é aceitável, perda não

---

7. Integração com Lambda

7.1 Modelo recomendado

- Consumer chama Lambda **sincronamente**
- Payload contém:
    - Lista de mensagens
    - Metadados (topic, partition, offset)

```
{
  "topic": "mt-c400",
  "partition": 3,
  "messages": [...]
}
```

7.2 Por que não assíncrono?

- Você perde controle de sucesso/falha
- Não sabe quando commitar offsets
- Quebra consistência

---

8. Impacto do Aumento de Partições no Consumer

8.1 O que acontece tecnicamente?

Quando você aumenta partições:

1. Kafka dispara **rebalance**
2. Consumers pausam
3. Partições são redistribuídas
4. Consumo continua

8.2 Impactos práticos

Impacto	Explicação
Rebalance	Pausa temporária
Ordem	Ordem global quebra
Throughput	Aumenta potencial
Código	Não muda (se bem feito)

👉 **Consumer bem feito não precisa mudar uma linha**

---

9. Pontos de Atenção (riscos reais)

9.1 Rebalance frequente

Causas:

- Muitas instâncias subindo/descendo
- Timeout mal configurado

Mitigação:

- Ajustar:
    - `session.timeout.ms`
    - `max.poll.interval.ms`

9.2 Backpressure

Se Lambda ficar lenta:

- Batch cresce
- Lag aumenta
- ASG tenta escalar

Mitigação:

- Timeout de Lambda
- Circuit breaker
- Limite de batch

9.3 Lag invisível

Se você não monitora:

- O sistema "parece vivo"
- Mas está horas atrasado

Obrigatório:

- Monitorar **Consumer Group Lag**

---

10. Perguntas duras que vão surgir (e respostas)

❓ "E se dobrar o volume em 6 meses?"

✔ Resposta:

> Aumentamos partições, o consumer absorve automaticamente via rebalance. O código já está preparado.

❓ "E se uma EC2 cair?"

✔ Resposta:

> Kafka redistribui partições para os outros consumers automaticamente.

❓ “Por que não Lambda direto consumindo Kafka?”

✔ Resposta:

> Lambda não é ideal para:

- Long-lived connections
- Alto throughput contínuo
- Controle fino de offset

---

11. Checklist de POC – Consumer

Código

- ☐ Consumer Group configurado
- ☐ Commit manual
- ☐ Batch size + timeout
- ☐ Logs estruturados

Infra

- ☐ ASG configurado
- ☐ IAM para Lambda invoke
- ☐ Certificados mTLS

Observabilidade

- ☐ Lag por consumer group
- ☐ Tempo de batch
- ☐ Erros de Lambda

---

12. Conclusão (frase de arquiteto)

> **O consumer é o elemento que transforma paralelismo do Kafka em capacidade real de processamento.**
> Se ele for stateless, idempotente e orientado a consumer groups, a arquitetura escala horizontalmente sem reescrita, mesmo com aumento de partições ao longo do tempo.

Perfeito. Abaixo está **um diagrama Mermaid focado exclusivamente no CONSUMER (.NET em EC2)**, cobrindo:

- Consumer Group
- Atribuição de partições
- Batching
- Commit de offset
- Chamada ao Lambda
- Rebalance

- Observabilidade

Você pode colar isso **diretamente no draw.io (diagrams.net)** ou em qualquer editor que suporte Mermaid.

📊 Diagrama Mermaid — Consumer Kafka (.NET + EC2)

```
flowchart TD
    subgraph MSK["Amazon MSK (Kafka Cluster)"]
        T1["Tópico: mt-c400"]
        P0["Partição 0"]
        P1["Partição 1"]
        P2["Partição 2"]
        Pn["Partição N"]

        T1 --> P0
        T1 --> P1
        T1 --> P2
        T1 --> Pn
    end

    subgraph CG["Consumer Group: mt-c400-consumer-group"]
        C1["EC2 Consumer #1 (.NET)"]
        C2["EC2 Consumer #2 (.NET)"]
        C3["EC2 Consumer #3 (.NET)"]
    end

    P0 -->|Assign| C1
    P1 -->|Assign| C2
    P2 -->|Assign| C3
    Pn -->|Assign| C1

    subgraph C1_FLOW["Fluxo Interno do Consumer"]
        POLL["Poll() Kafka"]
        BUFFER["Buffer de Mensagens"]
        CHECK{"Batch Size\nou Timeout?"}
        INVOKE["Invoke Lambda\n(Sync)"]
        COMMIT["Commit Offset"]
    end

    C1 --> POLL
    POLL --> BUFFER
    BUFFER --> CHECK
    CHECK -- Não --> POLL
    CHECK -- Sim --> INVOKE
    INVOKE -->|OK| COMMIT
    COMMIT --> POLL

    INVOKE -->|Erro| DLQ["Retry / DLQ / Backoff"]
```

```
    subgraph OBS["Observabilidade"]
        LAG["Consumer Group Lag"]
        METRICS["CPU / Memória"]
        LOGS["Logs Estruturados"]
    end

    C1 --> LAG
    C2 --> LAG
    C3 --> LAG

    C1 --> METRICS
    C2 --> METRICS
    C3 --> METRICS

    C1 --> LOGS
    C2 --> LOGS
    C3 --> LOGS

    subgraph REBALANCE["Evento de Rebalance"]
        SCALE["Scale Out / In EC2"]
        REDIST["Redistribuição de Partições"]
    end

    SCALE --> REDIST
    REDIST --> C1
    REDIST --> C2
    REDIST --> C3
```

🧠 Como interpretar esse diagrama (para explicar aos pares)

1️⃣ MSK e Partições

- O tópico `mt-c400` é dividido em **N partições**
- Kafka garante **ordem por partição**

2️⃣ Consumer Group

- Todas as EC2 usam o **mesmo group.id**
- Kafka atribui partições dinamicamente
- **1 partição → 1 consumer ativo**

3️⃣ Fluxo interno do Consumer

- `Poll()` lê mensagens
- Mensagens vão para buffer
- Quando:

- Batch size **OU**
- Timeout
  → Lambda é invocado

4️⃣ Commit de Offset

- Só ocorre **após Lambda responder OK**
- Garante **at-least-once**

5️⃣ Rebalance

- Acontece quando:
  - Nova EC2 sobe
  - EC2 cai
  - Partições aumentam
- Kafka redistribui automaticamente

6️⃣ Observabilidade

- **Lag** é o principal KPI
- CPU/Memória mostram saturação
- Logs ajudam a diagnosticar rebalance e falhas

🎯 Mensagem-chave para a documentação

> O consumer é stateless, orientado a consumer group e desacoplado do número de partições. Isso permite escalar horizontalmente, absorver aumento de partições e manter confiabilidade sem mudanças de código.

Excelente pedido — **Consumer Group é o coração da sua arquitetura** e normalmente é onde arquitetos, SREs e seniors mais questionam.
Vou fazer em **três níveis**:

1. **Explicação conceitual profunda (o "porquê")**
2. **Como isso se traduz na sua aplicação .NET em EC2**
3. **Diagrama Mermaid didático para documentação e apresentação**

1️⃣ O que é um Consumer Group (conceito profundo)

Um **Consumer Group** no Kafka é um **mecanismo de coordenação distribuída** que resolve **três problemas fundamentais**:

🔷 Problema 1 – Escalabilidade horizontal

Sem consumer group:

- Um consumer lê tudo
- Limite físico de CPU / rede

Com consumer group:

- Várias instâncias leem **em paralelo**
- Kafka distribui as **partições** entre elas

> 📌 **Regra de ouro**
>
> - 1 partição → **somente 1 consumer ativo**
> - 1 consumer → **pode ler várias partições**

🔷 Problema 2 – Garantia de ordem

Kafka **não garante ordem global**, apenas:

> **Ordem dentro da mesma partição**

O consumer group **preserva essa garantia**, porque:

- Uma partição nunca é processada por dois consumers ao mesmo tempo
- Logo, a sequência de mensagens é mantida

🔷 Problema 3 – Tolerância a falhas

Se um consumer cai:

- Kafka detecta falha via **heartbeat**
- Redistribui as partições automaticamente
- Outro consumer assume

> Isso é **self-healing**, sem intervenção humana.

2️⃣ Como funciona internamente (mecanismo real)

Componentes envolvidos:

Componente	Função
Group Coordinator	Broker Kafka responsável pelo grupo
Heartbeat	Sinal periódico de vida do consumer

Componente	Função
Partition Assignment	Algoritmo de distribuição
Offsets	Posição de leitura por partição

🔄 Ciclo de vida de um Consumer Group

1. Consumer inicia
2. Envia `JoinGroup`
3. Recebe partições
4. Começa a consumir
5. Envia heartbeats
6. Commit de offsets
7. Continua até morrer ou escalar

⚠️ Rebalance (ponto crítico)

Rebalance ocorre quando:

- Nova EC2 sobe
- EC2 cai
- Partições aumentam
- Consumer fica lento

Durante o rebalance:

- Consumo **pausa**
- Partições são redistribuídas
- Offsets são reatribuídos

> ⚠️ Quanto mais partições e mais consumers, mais caro o rebalance.

3️⃣ Por que sua aplicação PRECISA definir um Consumer Group

Na sua arquitetura:

- EC2 Consumer é **stateless**
- Lambda recebe batches
- Kafka mantém estado (offsets)

👉 **O estado não está na aplicação, está no Kafka**

Isso permite:

- Escalar EC2 sem reconfiguração
- Substituir instâncias sem perda
- Deploy seguro (rolling update)

🔐 group.id = contrato operacional

O `group.id` define:

- Quem divide carga
- Quem compete pelas mensagens
- Quem mantém offset

> Se mudar o `group.id`, Kafka entende como **aplicação nova**

4️⃣ Consumer Group no contexto .NET + MSK

Configuração típica (.NET)

```
var config = new ConsumerConfig
{
    BootstrapServers = "...",
    GroupId = "mt-c400-consumer-group",
    EnableAutoCommit = false,
    AutoOffsetReset = AutoOffsetReset.Earliest,
    MaxPollIntervalMs = 300000,
    SessionTimeoutMs = 45000
};
```

O que isso significa arquiteturalmente

Config	Impacto
`GroupId`	Define cluster lógico de consumers
`EnableAutoCommit=false`	Controle transacional
`MaxPollIntervalMs`	Protege contra consumer lento
`SessionTimeoutMs`	Define sensibilidade a falhas

5️⃣ Consumer Group ≠ Tópico ≠ Aplicação

Essa confusão é comum:

Conceito	Significado
Tópico	Fluxo de dados
Consumer Group	Cluster lógico de leitura
Consumer	Instância física
Aplicação	Código que implementa o consumer

6️⃣ Diagrama Mermaid — Consumer Group em detalhes

```
flowchart LR
    subgraph Kafka["Amazon MSK (Kafka)"]
        T["Tópico: mt-c400"]
        P0["Partição 0"]
        P1["Partição 1"]
        P2["Partição 2"]
        P3["Partição 3"]

        T --> P0
        T --> P1
        T --> P2
        T --> P3
    end

    subgraph CG["Consumer Group: mt-c400-consumer-group"]
        C1["Consumer EC2 #1"]
        C2["Consumer EC2 #2"]
        C3["Consumer EC2 #3"]
    end

    P0 -->|assign| C1
    P1 -->|assign| C2
    P2 -->|assign| C3
    P3 -->|assign| C1

    C1 -->|heartbeat| Kafka
    C2 -->|heartbeat| Kafka
    C3 -->|heartbeat| Kafka

    C1 -->|commit offset| Kafka
    C2 -->|commit offset| Kafka
    C3 -->|commit offset| Kafka

    subgraph Rebalance["Rebalance Event"]
        SCALE["Scale Out / Failure"]
        REDIST["Reatribuição de Partições"]
```

```
    end

    SCALE --> REDIST
    REDIST --> CG
```

7️⃣ Argumento forte para convencer superiores

> O Consumer Group transforma múltiplas instâncias EC2 em **um único consumidor lógico**, com:
>
> - Escalabilidade horizontal
> - Tolerância a falhas
> - Preservação de ordem
> - Controle de estado centralizado
>
> Sem ele, a arquitetura seria rígida, frágil e cara.

8️⃣ Riscos e cuidados (pontos que seniors vão perguntar)

Risco	Mitigação
Rebalance frequente	Menos consumers que partições
Consumer lento	Batching + async
Lag alto	Auto scaling
Lambda lento	Backpressure
Commit incorreto	Manual commit

✅ Frase de encerramento para a documentação

> O Consumer Group é o mecanismo que permite escalar consumo, garantir ordem e tolerar falhas sem adicionar estado ou complexidade ao serviço.

😈 Principal Engineer (abre a reunião)

> **“Vamos direto ao ponto. Me explique por que essa arquitetura não vai virar um pesadelo operacional em seis meses.”**

✅ Sua resposta (calma, técnica):

> Porque ela se apoia em primitivas maduras do Kafka:

> - consumer group para coordenação
> - partições como unidade de paralelismo
> - estado externo (offsets)
>
> Não criamos lógica distribuída customizada. Delegamos complexidade ao Kafka, que já resolve isso há anos.

😈 SRE (já atacando)

> **"Rebalance. Toda vez que eu escuto essa palavra, alguém está escondendo downtime. Quanto tempo seu sistema fica parado?"**

✅ Resposta certa (sem mentir):

> O consumo pausa durante o rebalance, sim.
>
> Mas:
>
> - não perdemos mensagens
> - o backlog é absorvido pelas partições
> - usamos batch para amortecer
>
> É um pause controlado, não downtime funcional.

> E evitamos rebalance frequente controlando autoscaling.

😈 SRE (mais agressivo)

> **"Quantos rebalances você espera por dia?"**

❌ Resposta errada:

> "Depende..."

✅ Resposta correta:

> Em steady state: zero.
>
> Apenas em:
>
> - deploy
> - falha de instância
> - scaling manual ou automatizado
>
> Não é um evento de runtime normal.

😈 Principal Engineer (olhando o desenho)

> **"Você tem 40 partições e 10 consumers. Por quê?"**

✅ Resposta forte:

> Porque partições são capacidade futura.
>
> Consumers são capacidade atual.
>
> Partições a mais:
>
> - reduzem impacto de scaling
> - evitam redistribuição estrutural
> - permitem crescimento sem mudança de MSK

😈 FinOps (entra pesado)

> **"Você está criando EC2 só para chamar Lambda. Isso é burrice ou luxo?"**

😐 silêncio na call...

✅ Resposta que salva:

> Lambda não é um bom consumer Kafka com mTLS e controle fino de batch.
>
> EC2:
>
> - controla backpressure
> - controla commit
> - reduz invocações Lambda via batch
>
> No fim, reduz custo total e risco operacional.

😈 FinOps (pressionando)

> **"Quanto custa esse batch?"**

✅ Resposta madura:

> Batch é uma decisão econômica e técnica.
>
> Menos invocações Lambda
> Menos custo por request
> Melhor uso de CPU
>
> O batch é parametrizável e monitorado via latency e lag.

😈 Principal Engineer (ataque clássico)

> **"Por que não um tópico só? Vocês estão complicando demais."**

❌ Resposta fraca:

> “Porque é melhor…”

✅ Resposta correta:

> Um tópico único mistura domínios, prioridades e ritmos.
>
> Isso gera:
>
> - vizinho barulhento
> - desperdício de CPU
> - blast radius alto
>
> Preferimos isolamento lógico para preservar estabilidade.

😈 SRE (cutucando ferida)

> **“E se um consumer ficar lento?”**

✅ Resposta certa:

> Kafka para de aceitar heartbeat.
>
> Dispara rebalance.
>
> Outro consumer assume.
>
> Offset só avança se o Lambda responder OK.

😈 SRE (quase gritando)

> **“E SE O LAMBDA FICAR LENTO?”**

✅ Resposta que demonstra senioridade:

> O consumer aplica backpressure automaticamente.
>
> Lag sobe.
>
> Observabilidade detecta.
>
> Scaling ou mitigação é acionada.
>
> Nenhuma mensagem é perdida.

😈 Principal Engineer (olhar mortal)

> **“O que quebra quando você aumenta partições?”**

✅ Resposta honesta (não tente esconder):

> A ordenação por chave é afetada.
>
> Por isso:
>
> - fazemos isso de forma controlada
> - temos playbook
> - só aumentamos quando o ganho supera esse impacto
>
> É um trade-off consciente.

😈 Principal Engineer (final)

> **“Qual é o maior risco dessa arquitetura?”**

❌ Resposta errada:

> “Não tem.”

✅ Resposta correta:

> Rebalance excessivo e downstream imprevisível.
>
> Ambos são mitigados com:
>
> - observabilidade
> - batch
> - controle de scaling
>
> São riscos conhecidos, não surpresas.

😈 Pergunta final (silêncio total)

> **“Se isso der errado, quem vai acordar às 3 da manhã?”**

✅ Resposta de quem passa:

> A arquitetura foi pensada para que falhas sejam eventos normais e observáveis.
>
> O sistema degrada com lag, não com perda.
>
> Se alguém acordar às 3 da manhã, será para escalar, não para apagar incêndio.

🏁 Resultado da reunião

👉 **Aprovado com ressalvas**
👉 Pedido de:

- métricas claras de lag

- limites de batch
- playbook de rebalance
- simulação em POC

🎯 Dica final (importante)

Em review hostil, **não tente vender perfeição**.
Venda:

- trade-offs conscientes
- riscos conhecidos
- mecanismos de mitigação

Isso é o que diferencia **arquiteto experiente** de “desenhista de diagrama”.

📄 POST-MORTEM TÉCNICO

Incidente: Aumento de Lag e Atraso no Processamento
Sistema: MSK + Consumer Group (.NET EC2) + Lambda
Severidade: SEV-2
Duração: 47 minutos
Impacto: Atraso no processamento downstream (sem perda de dados)

1️⃣ Resumo Executivo

Entre **10:13 e 11:00**, o sistema apresentou um aumento significativo de **consumer lag** no tópico `mt-c400`, atingindo um pico de **~1,2 milhão de mensagens pendentes**.

Nenhuma mensagem foi perdida.
O sistema se recuperou automaticamente após mitigação operacional.

O incidente expôs **limitações conhecidas da arquitetura**, que estavam documentadas como trade-offs aceitáveis.

2️⃣ Linha do Tempo (Timeline)

Horário	Evento
10:13	Deploy de nova versão do consumer
10:14	Início de rebalance
10:15	Lambda começa a responder mais lento
10:18	Lag cresce rapidamente

Horário	Evento
10:25	Alerta de lag > threshold
10:30	Autoscaling acionado
10:42	Rebalance completo
11:00	Lag normalizado

3️⃣ Impacto ao Negócio

- ❌ **Nenhuma perda de dados**
- ⚠️ Atraso de até **8 minutos** no processamento downstream
- ✅ Eventos críticos continuaram sendo ingeridos
- ❌ SLA de latência temporariamente violado

4️⃣ O que aconteceu (Análise Técnica Profunda)

🔷 4.1 Rebalance em cascata

Durante o deploy:

- 3 instâncias EC2 foram reiniciadas
- Kafka detectou perda de heartbeat
- Rebalance foi disparado

⚠️ **Importante:**
O rebalance pausou temporariamente o consumo **por design**.

🔷 4.2 Lambda como gargalo downstream

Durante o rebalance:

- Batch acumulou mensagens
- Lambda começou a responder mais lentamente
- Tempo médio de resposta subiu de 120ms → 950ms

Isso causou:

- Atraso no commit de offsets
- Lag crescente

🔷 4.3 Por que o sistema não quebrou?

Porque:

- Commit era manual

- Offsets não avançaram incorretamente
- Kafka reteve mensagens
- Consumer Group garantiu reassignment correto

👉 **At-least-once garantido**

5️⃣ O que NÃO aconteceu (mitos comuns)

Medo comum	O que realmente aconteceu
“Perdemos mensagens”	❌ Kafka reteve tudo
“Consumers processaram duplicado”	⚠️ Pequena duplicação aceitável
“Sistema caiu”	❌ Sistema degradou
“Precisamos refatorar tudo”	❌ Arquitetura se comportou como esperado

---

6️⃣ Root Cause (Causa Raiz)

> **Causa primária:**
> Rebalance simultâneo + aumento inesperado de latência do Lambda.

> **Causa secundária:**
> Deploy sem estratégia de rolling update com limitação de instâncias simultâneas.

7️⃣ Decisões Arquiteturais que se provaram corretas

✅ Consumer Group

- Redistribuição automática
- Nenhuma intervenção manual
- Recuperação automática

✅ Batch controlado

- Reduziu custo Lambda
- Amorteceu pico de lag

✅ Commit manual

- Nenhuma perda
- Nenhum offset corrompido

8️⃣ Trade-offs que se manifestaram (e eram conhecidos)

⚠️ Rebalance pausa consumo

- Documentado
- Esperado
- Impacto temporário

⚠ Dependência de downstream

- Lambda lento impacta lag
- Mitigado por backpressure

9️⃣ O que poderia ter sido pior (e não foi)

- ❌ Commit automático → perda de mensagens
- ❌ Lambda direto no MSK → falhas imprevisíveis
- ❌ Estado no consumer → recuperação manual

1️⃣0️⃣ Ações Corretivas (Action Items)

🛠️ Curto Prazo

- Limitar deploy a 1 EC2 por vez
- Aumentar timeout do Lambda
- Ajustar batch size dinâmico

🛠️ Médio Prazo

- Separar tópicos por prioridade
- Criar Lambda dedicado para eventos críticos
- Ajustar autoscaling por lag

🛠️ Longo Prazo

- Consumer dedicado para high-priority
- Circuit breaker no downstream
- Teste de caos (rebalance forçado)

1️⃣1️⃣ Lições Aprendidas (o ponto mais importante)

> A arquitetura **não falhou**.
> Ela **degradou de forma previsível**.

Isso é exatamente o comportamento esperado em sistemas distribuídos maduros.

1️⃣2️⃣ Perguntas difíceis que surgiram no post-mortem

❓ “Isso vai acontecer de novo?”

> Sim. Rebalances são inevitáveis.

❓ “Podemos evitar totalmente?”

> Não, mas podemos **reduzir frequência e impacto**.

❓ “Vale a pena essa complexidade?”

> Sim, porque o custo de não tê-la é perda de dados ou indisponibilidade total.

1️⃣3️⃣ Diagrama Mermaid — Incidente e Recuperação

```
sequenceDiagram
    participant EC2 as Consumer EC2
    participant Kafka as MSK
    participant Lambda as Lambda
    participant SRE as SRE

    EC2->>Kafka: Poll()
    Kafka-->>EC2: Mensagens
    EC2->>Lambda: Invoke(batch)
    Lambda-->>EC2: Latência ↑
    EC2--xKafka: Commit atrasado
    Kafka->>Kafka: Lag cresce
    SRE->>EC2: Escala consumidores
    Kafka->>EC2: Rebalance
    EC2->>Kafka: Consumo retomado
    EC2->>Lambda: Processamento normal
```

1️⃣4️⃣ Frase final para fechar o post-mortem

> Em sistemas distribuídos, falha não é exceção — é estado transitório.
> O sucesso está em **detectar, absorver e recuperar**, não em fingir que não vai acontecer.

Perfeito.
Abaixo está **uma explicação EXTREMAMENTE detalhada**, acompanhada de **diagramas Mermaid grandes**, cobrindo **exatamente o que você tem que fazer**, **como**, **por quê** e **como tudo se conecta**, respeitando **todas as restrições**:

- MSK em **outra conta**
- Sem acesso administrativo ao cluster
- Apenas **criação/configuração de tópicos**
- Producer e Consumer em **EC2 (.NET)**
- Consumer chamando **Lambda distribuidor**
- Uso explícito de **tópicos, partições e consumer group**

Vou dividir em **4 blocos grandes** para facilitar documentação e defesa arquitetural.

🗃️ VISÃO GERAL DA ARQUITETURA (O TODO)

Antes de entrar em Producer/Consumer, é essencial **fixar o modelo mental correto**:

> **Kafka (MSK) é o sistema de coordenação e estado**
> Producer e Consumer são **stateless**
> Consumer Group é o **mecanismo de escala e resiliência**

📌 Arquitetura lógica (alto nível)

```
flowchart LR
    subgraph AccountA["Conta A – Infra Kafka"]
        MSK["Amazon MSK\n(Kafka Cluster)"]
        T1["Tópico mt-c400\n(20+ Partições)"]
        T2["Tópico mt-c300\n(8+ Partições)"]
        MSK --> T1
        MSK --> T2
    end

    subgraph AccountB["Conta B – Aplicações"]
        P["Producer .NET\n(EC2)"]
        C["Consumer .NET\n(EC2)"]
        L["Lambda\nDistribuidor"]
    end

    P -->|Produce| MSK
    MSK -->|Consume| C
    C -->|Batch| L
```

🧩 BLOCO 1 — MODELAGEM DE TÓPICOS E PARTIÇÕES (DECISÃO CHAVE)

🎯 Objetivo

- Permitir **13k msg/s**
- Isolar domínios
- Minimizar rebalance
- Preparar crescimento

🔷 Estrutura proposta de tópicos

Tópico	Função	Partições	Observação
`mt-c400`	Domínio mais quente (80%)	20–40	Alta paralelização
`mt-c300`	Domínio médio	8–12	Menor volume
Outros	Específicos	conforme	Isolamento

> ⚠ Importante:
> **Códigos NÃO viram tópicos**, eles viram **chaves de particionamento**

🔷 Regra de ouro

> **Tópicos = contratos**
> **Partições = paralelismo**
> **Chave = ordem**

🧰 BLOCO 2 — PRODUCER .NET (EC2)

🎯 Papel do Producer

- Receber mensagens de várias fontes
- Decidir:
  - **Qual tópico**
  - **Qual chave**
- Nunca decidir consumer, partição manual ou offset

🔁 Fluxo completo do Producer

```
flowchart TD
    START["Start Producer Service"]
    LOADCFG["Load Config\n(Tópicos, Brokers, TLS)"]
    TLS["Load Certificado\n(mTLS via S3)"]
    INIT["Init Kafka Producer\n(.NET)"]

    LOOP["Loop de Produção"]
    BUILD["Build Mensagem JSON"]
    SELECT["Selecionar Tópico\n(mt-c400 / mt-c300)"]
    KEY["Definir Message Key\n(código / deviceId)"]
    SEND["ProduceAsync()"]
    ACK{"ACK OK?"}
    RETRY["Retry / DLQ"]
    METRIC["Emitir Métricas"]
```

```
    START --> LOADCFG --> TLS --> INIT --> LOOP
    LOOP --> BUILD --> SELECT --> KEY --> SEND --> ACK
    ACK -- Sim --> METRIC --> LOOP
    ACK -- Não --> RETRY --> LOOP
```

🔷 Como o Producer decide partição (conceito crítico)

❌ O que NÃO fazer

- Não escolher partição manualmente
- Não tentar “balancear na mão”
- Não criar lógica distribuída customizada

✅ O que fazer

- **Definir a `Message.Key`**
- Kafka decide a partição via **hash da key**

```
var message = new Message<string, string>
{
    Key = codigoOuDeviceId,
    Value = json
};

await producer.ProduceAsync("mt-c400", message);
```

🔷 Por que isso é correto?

- Garante ordem por chave
- Permite aumento de partições
- Mantém producer simples
- Evita dependência do MSK

⚠ Pontos perigosos no Producer

Risco	Mitigação
Hot partition	Escolher chave bem distribuída
Burst de produção	Buffer interno do Kafka
Falha de broker	ACK + retry
Certificado vencido	Rotação via S3

📦 BLOCO 3 — CONSUMER .NET + CONSUMER GROUP

🎯 Papel do Consumer

- Ler em paralelo
- Preservar ordem por partição
- Batching
- Chamar Lambda
- Commit manual

🔷 Consumer Group como unidade lógica

```
flowchart LR
    subgraph Kafka["MSK"]
        T["Tópico mt-c400"]
        P0["P0"]
        P1["P1"]
        P2["P2"]
        P3["P3"]
        T --> P0 --> P1 --> P2 --> P3
    end

    subgraph CG["Consumer Group: mt-c400-group"]
        C1["EC2 Consumer #1"]
        C2["EC2 Consumer #2"]
        C3["EC2 Consumer #3"]
    end

    P0 --> C1
    P1 --> C2
    P2 --> C3
    P3 --> C1
```

> Kafka garante:
> **1 partição → 1 consumer ativo**

🔁 Fluxo interno do Consumer

```
flowchart TD
    START["Start Consumer"]
    LOADCFG["Load group.id"]
    TLS["Load mTLS Cert"]
    SUB["Subscribe Topics"]
```

```
    POLL["Poll()"]
    BUFFER["Buffer Mensagens"]
    CHECK{"Batch\nou Timeout?"}
    CALL["Invoke Lambda"]
    COMMIT["Commit Offset"]
    ERROR["Retry / Backoff"]

    START --> LOADCFG --> TLS --> SUB --> POLL
    POLL --> BUFFER --> CHECK
    CHECK -- Não --> POLL
    CHECK -- Sim --> CALL
    CALL -- OK --> COMMIT --> POLL
    CALL -- Fail --> ERROR --> POLL
```

🔷 Por que usar Consumer Group

- Escala horizontal automática
- Rebalance em falha
- Estado fora da aplicação
- Deploy seguro

🧰 BLOCO 4 — LAMBDA DISTRIBUIDOR

🎯 Papel do Lambda

- NÃO consumir Kafka
- NÃO manter estado
- Apenas:
    - Roteamento
    - Fan-out
    - Enriquecimento

```
flowchart LR
    C["Consumer EC2"]
    L["Lambda Distribuidor"]
    S1["Serviço A"]
    S2["Serviço B"]
    S3["Serviço C"]

    C -->|Batch JSON| L
    L --> S1
```

```
    L --> S2
    L --> S3
```

⚠ Trade-off consciente

- Lambda lento → lag sobe
- Nenhuma perda
- Backpressure natural

🧠 AMARRAÇÃO FINAL (PARA DOCUMENTAÇÃO)

O que essa arquitetura FAZ bem

- Escala horizontal
- Tolera falhas
- Isola responsabilidades
- Permite crescimento sem refatoração

O que ela NÃO tenta fazer

- Ordem global
- Latência zero
- Processamento síncrono fim-a-fim

📌 Frase de ouro para fechar

> O Kafka é o cérebro do sistema.
> Producer injeta eventos.
> Consumer Group processa em paralelo.
> Lambda distribui.
>
> Nenhuma peça sabe demais sobre a outra.
