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 fundamentais 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.”