Documentação da Implementação do Ingestor

Sumário

[Entendimento da Proposta de Negócio 2](#_Toc194797164)

[Ingestão de Pulsos de Consumo: 2](#_Toc194797165)

[Acumulação e Agregação: 2](#_Toc194797166)

[Persistência e Disponibilização: 2](#_Toc194797167)

[Escalabilidade e Tolerância a Falhas: 2](#_Toc194797168)

[Integração com Outros Componentes: 2](#_Toc194797169)

[Resumo da Proposta 3](#_Toc194797170)

[Detalhamentos e Definições Técnicas 4](#_Toc194797171)

[Decisões Técnicas 4](#_Toc194797172)

[Linguagem e Framework: 4](#_Toc194797173)

[Processamento Assíncrono: 4](#_Toc194797174)

[Persistência e Agregação: 4](#_Toc194797175)

[Gerenciamento de Gerações: 4](#_Toc194797176)

[Disparo e Deleção: 5](#_Toc194797177)

[Monitoramento e Logging: 5](#_Toc194797178)

[Diagramas da Arquitetura 6](#_Toc194797179)

[Fluxo de Processamento 10](#_Toc194797180)

[Considerações para Produção 10](#_Toc194797181)

[Implementação do Ingestor 11](#_Toc194797182)

[Estrutura do Código 11](#_Toc194797183)

[Funcionalidades Implementadas 12](#_Toc194797184)

[API de Ingestão: 12](#_Toc194797185)

[Processamento Assíncrono: 12](#_Toc194797186)

[Persistência e Agregação: 12](#_Toc194797187)

[Gerações Alternadas: 12](#_Toc194797188)

[Disparo e Deleção: 12](#_Toc194797189)

[Monitoramento: 13](#_Toc194797190)

# Entendimento da Proposta de Negócio

**Características de uma Plataforma de Bilhetagem**

A proposta de negócio descreve uma plataforma de bilhetagem para medir, registrar, disponibilizar e cobrar o consumo de recursos dos clientes. Abaixo estão as características principais que a parte de ingestão de um sistema de bilhetagem deve ter, com base no enunciado:

## Ingestão de Pulsos de Consumo:

* + A plataforma deve receber "pulsos" de consumo, que representam unidades mínimas de uso (ex.: 307KB de download, 60 segundos de 1GB armazenado).
  + Cada pulso contém:
    - tenantId: Identificador do cliente (quem paga).
    - product\_sku: Identificador do produto utilizado (ex.: SKU-997).
    - used\_amount: Quantidade consumida (métrica numérica).
    - use\_unity: Unidade da métrica (ex.: "KB", "GB x seg").
  + O volume de pulsos é "gigantesco", exigindo alta escalabilidade e eficiência na ingestão.

## Acumulação e Agregação:

* + Os pulsos devem ser acumulados e agregados (ex.: somar o used\_amount para cada combinação de tenant, product\_sku, e use\_unity).

## Persistência e Disponibilização:

* + O armazenamento deve ser eficiente, mantendo apenas o total consumido até o momento para cada tenant, product\_sku, e use\_unity.

## Escalabilidade e Tolerância a Falhas:

* + A plataforma deve suportar um volume massivo de pulsos, indicando a necessidade de flexibilização de escalabilidade horizontal.
  + Como o sistema de bilhetagem é crítico para o negócio, ele deve ser robusto, evitando a perda de pulsos em caso de falhas.
  + Recomendável métricas e logs para eventuais consultas.

## Integração com Outros Componentes:

* + A plataforma interage com o sistema de Processador & Armazenador por envios de pacotes com os pulsos agregados.

## Resumo da Proposta

A Parte de um gestor de uma plataforma de bilhetagem da precisa ser um sistema escalável, robusto e eficiente, capaz de:

* Ingerir e agregar um volume massivo de pulsos de consumo.
* Manter dados temporariamente de forma segura e eficiente.
* Disponibilizar os dados para o sistema “Processador & Armazenador”.
* Garantir consistência e tolerância a falhas, já que a bilhetagem é crítica para a cobrança dos clientes.

# Detalhamentos e Definições Técnicas

## Decisões Técnicas

### Linguagem e Framework:

* + **Go** foi escolhido como linguagem devido à sua performance, concorrência nativa (goroutines e canais), e simplicidade.
  + **Gin** foi usado como framework HTTP para criar a API (POST /ingest).

### Processamento Assíncrono:

* + Uso de **canais (Go)** para desacoplar o recebimento dos pulsos do processamento.
  + Cada pulso recebido é enviado para um canal (s.pulses), e workers (goroutines) consomem os pulsos do canal para processá-los.
  + Um canal com buffer de 50.000 pulsos foi configurado para suportar picos de 2000 req/s por até 30 segundos.

### Persistência e Agregação:

* + Redis foi escolhido como banco de dados para persistência e agregação, devido à sua alta performance e suporte a operações atômicas.
  + O Redis terá réplicas e sentinelas para mitigação de indisponibilidade da instancia principal do Redis.
  + Cada pulso é armazenado no Redis com uma chave no formato generation:tenant:product\_sku:use\_unity.
  + O used\_amount é incrementado atomicamente usando IncrByFloat.

### Gerenciamento de Gerações:

* + Para evitar race conditions entre a leitura e a deleção dos pulsos, introduzi o conceito de gerações alternadas (Geração A e Geração B).
  + A geração atual é armazenada no Redis (chave current\_generation, valor "A" ou "B").
  + Ao iniciar o serviço do pulso, é armazenado o current\_generation proveniente do Redis dentro de uma variável do tipo atomic.value
  + A cada ciclo (hora em hora):
    - Processa os pulsos da geração atual e alterna no redis para quando os Workers forem atualizar a geração no valor atomic, eles estejam de acordo com a geração seguinte (ex.: se a geração atual é A, passa a ser B).
    - Isso garante que novos incrementos não sejam afetados pelo processamento pelo sender de pulsos antigos.

### Disparo e Deleção:

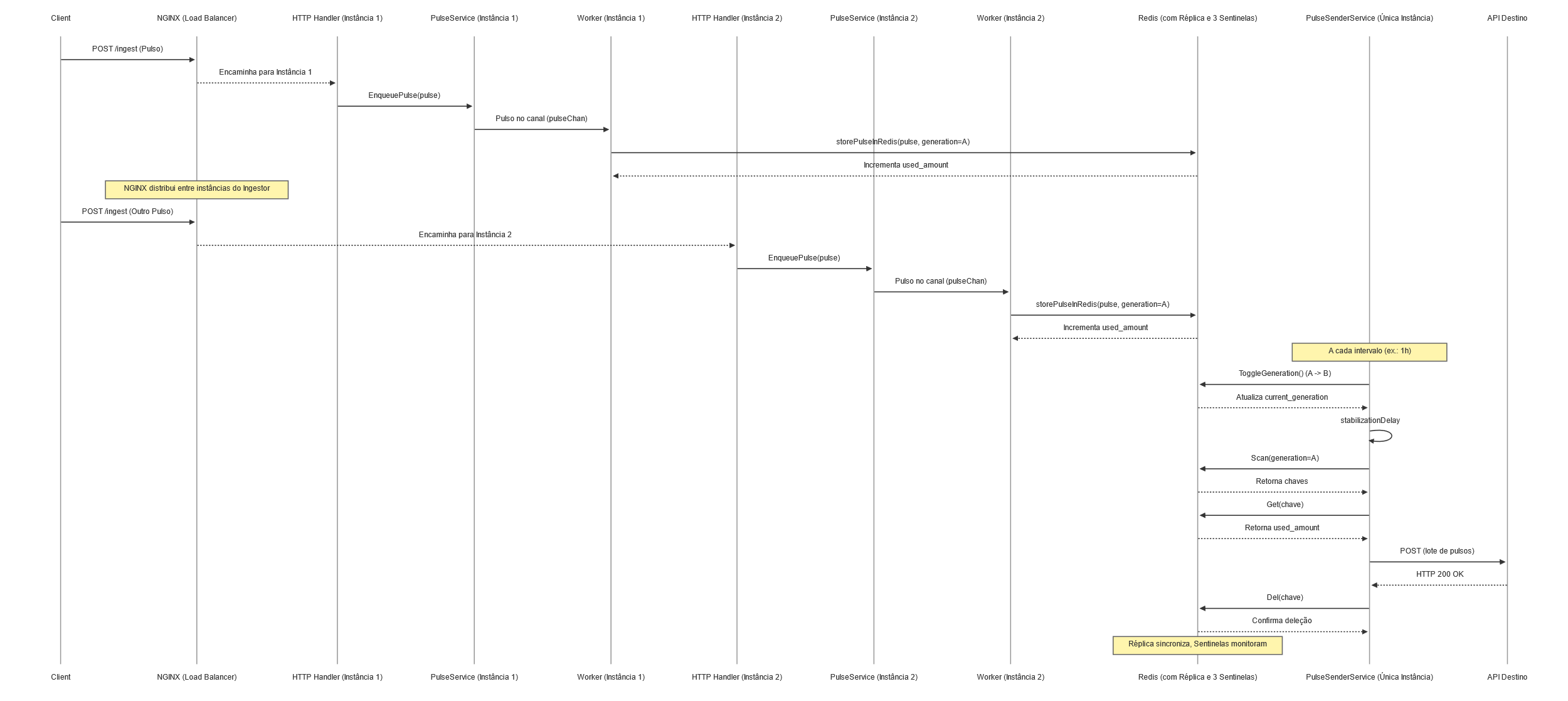
* + A cada hora, os pulsos da geração oposta são enviados em lotes para o Processador & Armazenador.
  + Em caso de sucesso, as chaves correspondentes são deletadas do Redis.
  + Em caso de falha, os pulsos são retidos para uma nova tentativa no próximo ciclo.

### Monitoramento e Logging:

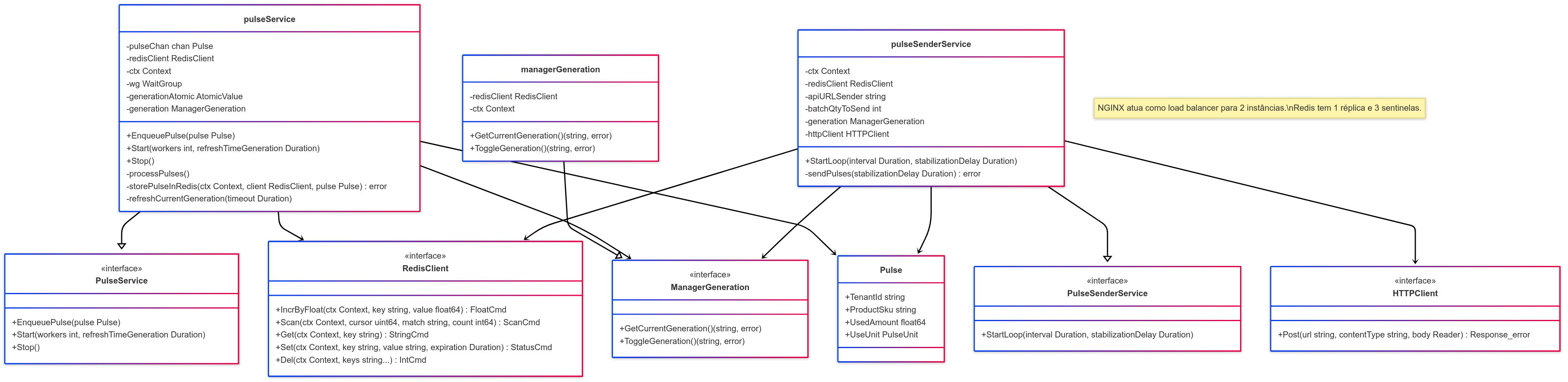
* + Prometheus foi usado para métricas (ex.: pulse\_channel\_size, pulses\_sent\_total, pulses\_sent\_failed\_total).
  + Zerolog foi usado para logging, com logs de nível Debug no console e Info em arquivo, com rotação via lumberjack.

## Diagramas da Arquitetura

* Diagrama de sequência:



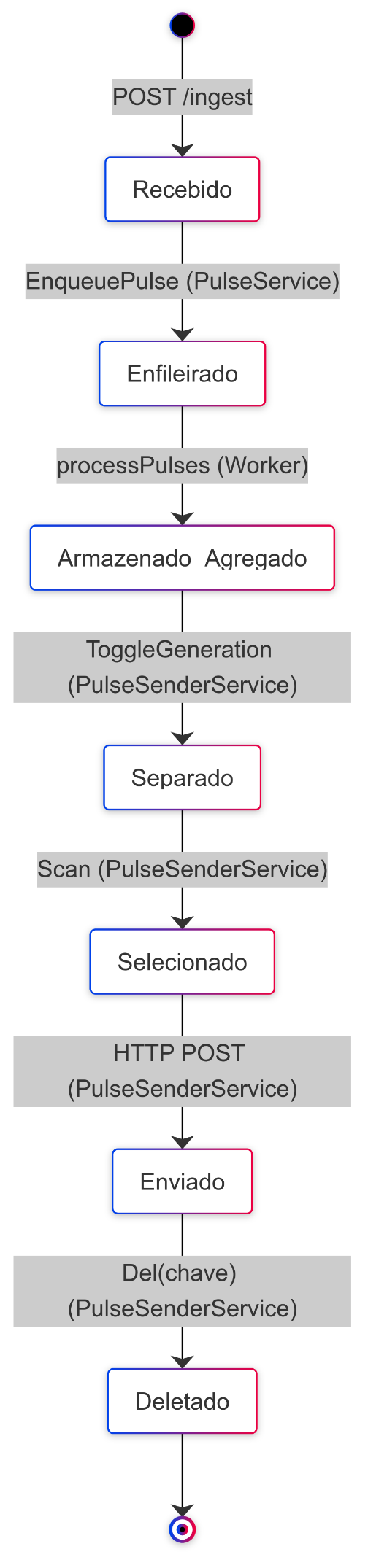
* Diagrama de classes:



* Diagrama de fluxo

Diagrama

O conteúdo gerado por IA pode estar incorreto.

* Diagrama e estados  
  

## Fluxo de Processamento

1. **Recebimento:** 
   * O cliente envia um pulso via POST /ingest.
   * O pulso é enviado para um canal (pulseChan).
2. **Processamento:** 
   * Workers consomem os pulsos do canal.
   * Cada pulso é armazenado no Redis com a geração atual (ex.: generation:A:tenant:tenant1:sku:SKU-1:useUnit:KB).
   * O used\_amount é incrementado atomicamente.
3. **Disparo:** 
   * A cada hora, os pulsos da geração até então atual são buscados, enviados em lotes, e deletados (se bem-sucedido).
   * A geração atual é alternada (ex.: de A para B).

# Implementação do Ingestor

## Estrutura do Código

O código foi implementado em Go e está organizado da seguinte forma:

* **cmd/ingestor/main.go:** Ponto de entrada do Ingestor.
* **build:** Pasta incluindo arquivos de build (Dockerfile, grafana, prometheus etc.)
* **cmd/producer/main.go:** Ponto de entrada do pulseProducer, usado para simular o envio de pulsos.
* **cmd/sender/main.go:** Ponto de entrada do pulseSender, usado para simular o envio de pulsos.
* **internal/clients/:** Utilitários para HTTP, logging e Redis.
* **internal/pulse/:** Lógica do Ingestor.
* **internal/pulseproducer/:** Lógica do pulseProducer (simulação de envio de pulsos).
* **log/:** Diretório para logs.
* **scripts/:** Scripts para executar o Ingestor e o pulseProducer.

## Funcionalidades Implementadas

### Infraestrutura:

* + Nginx para fazer o balanceamento entre as instancias definidas
  + Redis com 1 réplica e 3 sentinelas
  + Serviço para processamento (Ingestor)
  + Serviço para disparo e deleção (Sender)

### API de Ingestão:

* + Endpoint POST /ingest recebe pulsos no formato JSON
  + O pulso é enviado para um canal para processamento assíncrono.
  + Validação de payload
  + Documentação com swagger

### Processamento Assíncrono:

* + Um canal com buffer de 50.000 pulsos suporta picos de 2000 req/s por até 25 segundos.
  + 10 workers (goroutines) consomem os pulsos do canal e os armazenam no Redis de forma incremental.

### Persistência e Agregação:

* + Os pulsos são armazenados no Redis com chaves no formato generation:A:tenant:tenant1:sku:sku1:useUnit:KB.
  + Tolerância a falhas utilizando replica dos redis e sentinelas.
  + O used\_amount é incrementado atomicamente usando IncrByFloat do Redis.

### Gerações Alternadas:

* + A geração atual (A ou B) é armazenada no Redis (current\_generation) e em uma variável atômica no sistema do ingestor.
  + A cada hora, os pulsos da geração atual são processados, e a geração é alternada.

### Disparo e Deleção:

* + Os pulsos são enviados em lotes de 500 para o Processador & Armazenador a partir do sender.
  + Em caso de sucesso, as chaves correspondentes são deletadas.
  + Em caso de falha, os pulsos são retidos para serem enviados novamente no próximo ciclo.

### Monitoramento:

* + Métricas do Prometheus disponíveis em /metrics (ex.: tamanho do canal, pulsos enviados, falhas no envio).
  + Gráficos do Grafana para consumos das métricas na porta 9090.
  + Logs detalhados com zerolog gerenciados pelo lumberjack para consulta de depuração.