Relatório TP2

Diogo Oliveira Neiss - Pós graduação
https://github.com/diogoneiss/kubernetes-machine-learning-server

Arquivos

The ML code responsible for generating association rules and a version of your model:

./machine-learning

Code responsible for running the Web front-end

./rest_api

The client application, scripts, or Web front-end in charge of demonstrating access to your REST API.

Mapeamento de portas e rotas (se estiver fora do servidor será necessário um bind de portas para acessar o serviço)

Página html para testes: http://localhost:31000/
Frontend com swagger da api: http://localhost:31000/docs
Endpoint POST da api: http://localhost:31000/api/recommend/

The Dockerfile to build your containers in charge of running the REST API and the other responsible to run the model generation, together with any additional code required to build your container images.

Tudo está nas pastas de código, porém aqui estão os Dockerfiles.

./rest_api/Dockerfile
./machine-learning

The YAML files describing the Kubernetes deployment and service.

./kubernetes/deployment.yaml
./kubernetes/service.yaml
./kubernetes/pvc.yaml
./kubernetes/job.yaml

The YAML file describing the ArgoCD application. This file is called the "Manifest" in the Web interface. This can also be exported using the spec field/property in the output of argocd app get [appname] -o yaml.

./argocd_manifest

URLs e portas

Frontend de testes: http://localhost:31000/
Frontend com doc da api: http://localhost:31000/docs

Endpoint POST da api: http://localhost:31000/api/recommend/

Arquitetura da solução

Foi desenvolvido um serviço FastAPI vinculado ao PVC. Um arquivo é utilizado para polling de mudanças, permitindo verificar se as regras devem ser recarregadas periodicamente.

Para geração das regras foi feito um job simples, executado periodicamente. Cada execução usa um dataset diferente do utilizado na última vez, gera os arquivos necessários e grava no PVC.

Testes feitos e tempos - ArgoCD

Foram feitos testes envolvendo vários cenários. Uma variável de ambiente foi adicionada ao deployment para acionar restarts sempre que um push acontecesse.

Aumentar réplicas de 5 -> 12
ArgoCD demorou 55 segundos para perceber a mudança no yaml
Em 24 segundos todos os pods tinham sido criados

Trocar variável de ambiente no deployment
31 segundos para perceber a mudança no repositório
14 segundos para reiniciar todos os pods

Mudar a imagem de 12 réplicas
27 segundos para todos os pods serem reconstruídos e iniciados

Scale down 12 -> 3 sem mudança de imagem
6 segundos para remover os pods excedentes

Tempo de mudanças - CI/CD

Mudanças nos yaml demoraram em torno de 45 segundos para ir ao ar, sem downtime.

Como o frontend é eventualmente consistente, não há necessidade de parar a execução para substituir os dados ou algo do tipo. A substituição é extremamente rápida e os dados foram pré-processados o máximo possível pelo job de machine learning.

Foi observado que scale up/down também não teve muito impacto na execução, cada pod inicia extremamente rápido e o kubernetes é esperto o suficiente para não destinar tráfego para pods não inicializados completamente.

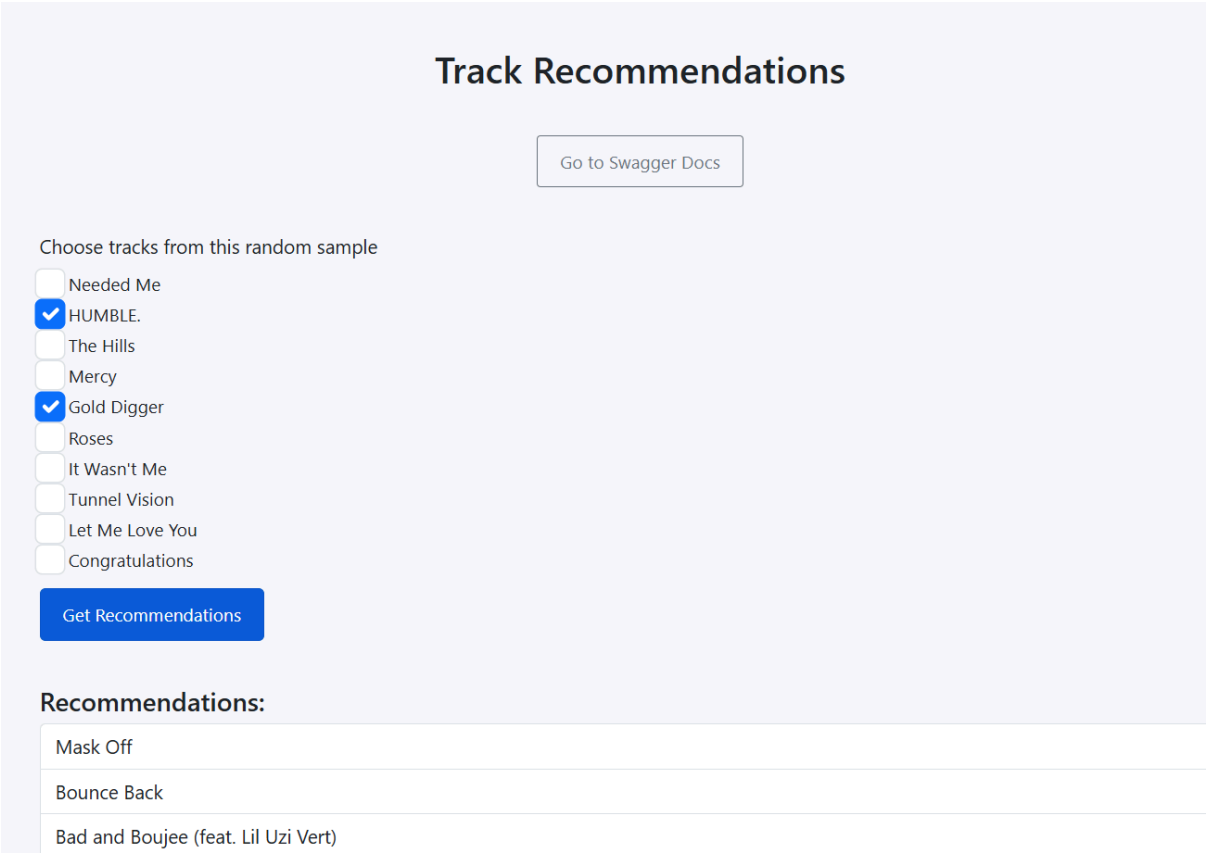
O fluxo completo de desenvolvimento, consistindo de alterar código nos 2 programas, buildar e enviar as imagens, e fazer um commit alterando os yamls para acionar o argocd demora em torno de 3 minutos, também sem downtime.

Alterações no dataset são eventuais por polling, então é necessário que o tempo adequado passe para que ele seja recarregado. Quando isso acontece é quase instantâneo, bastando carregar os arquivos pickle do PVC.

O job de machine learning demora em torno de 1 minuto para carregar o dataset adequado, gerar regras e gravar os pickles.

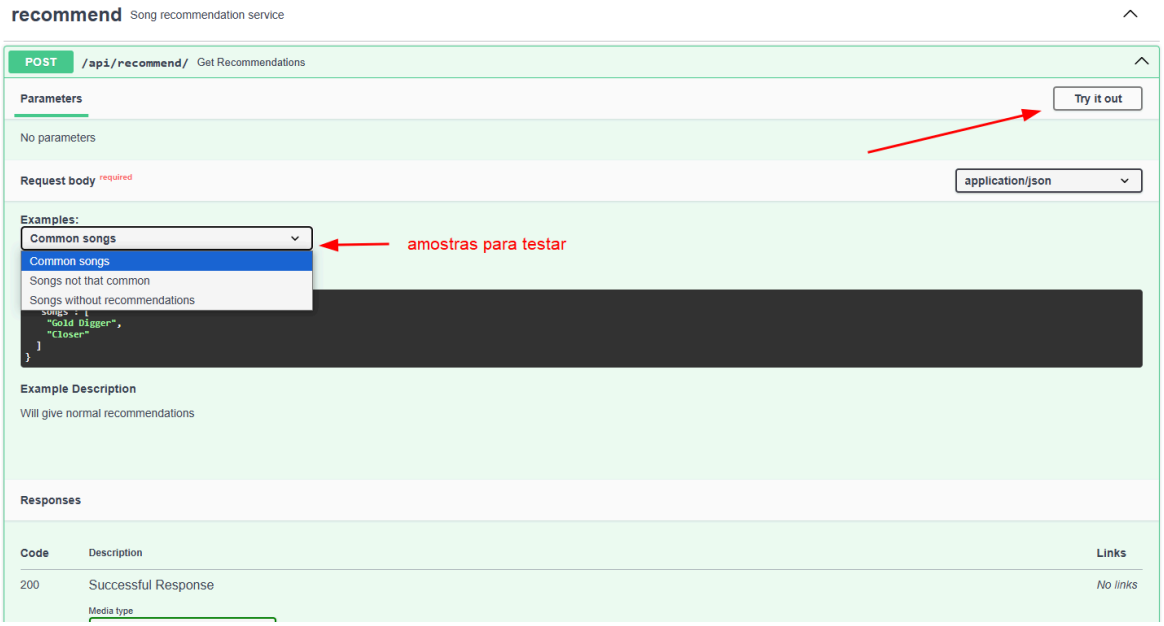
Frontend de testes

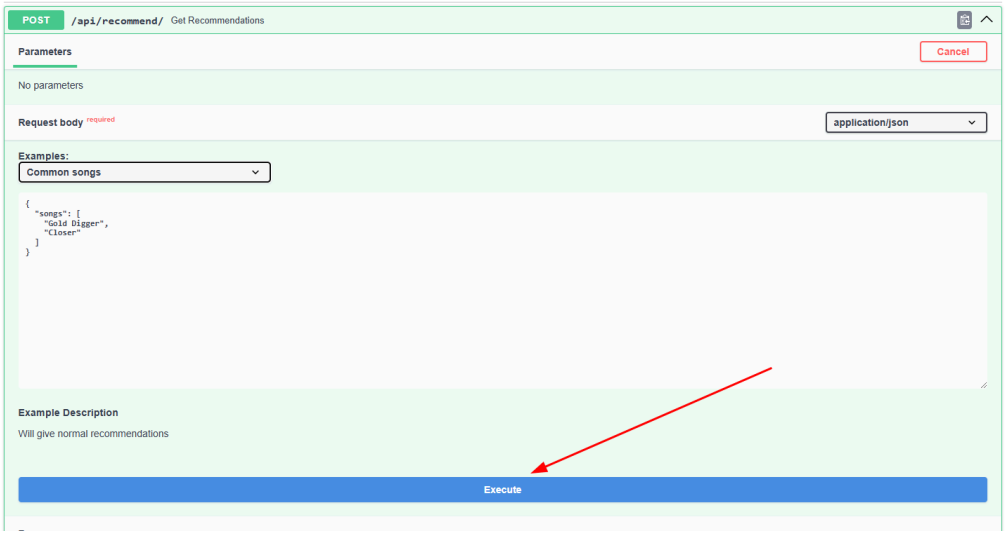
Criado frontend de testes disponível em “/”. Ele recupera uma amostra aleatória de músicas, cria um seletor de checklists e permite uma chamada para api.



Caso as músicas escolhidas sejam muito famosas, as recomendações ficarão enviesadas, já que são ordenadas pela correspondência de acordo com o conjunto inteiro solicitado

É possível também testar pelo swagger, disponível naquele botão ou em “/docs



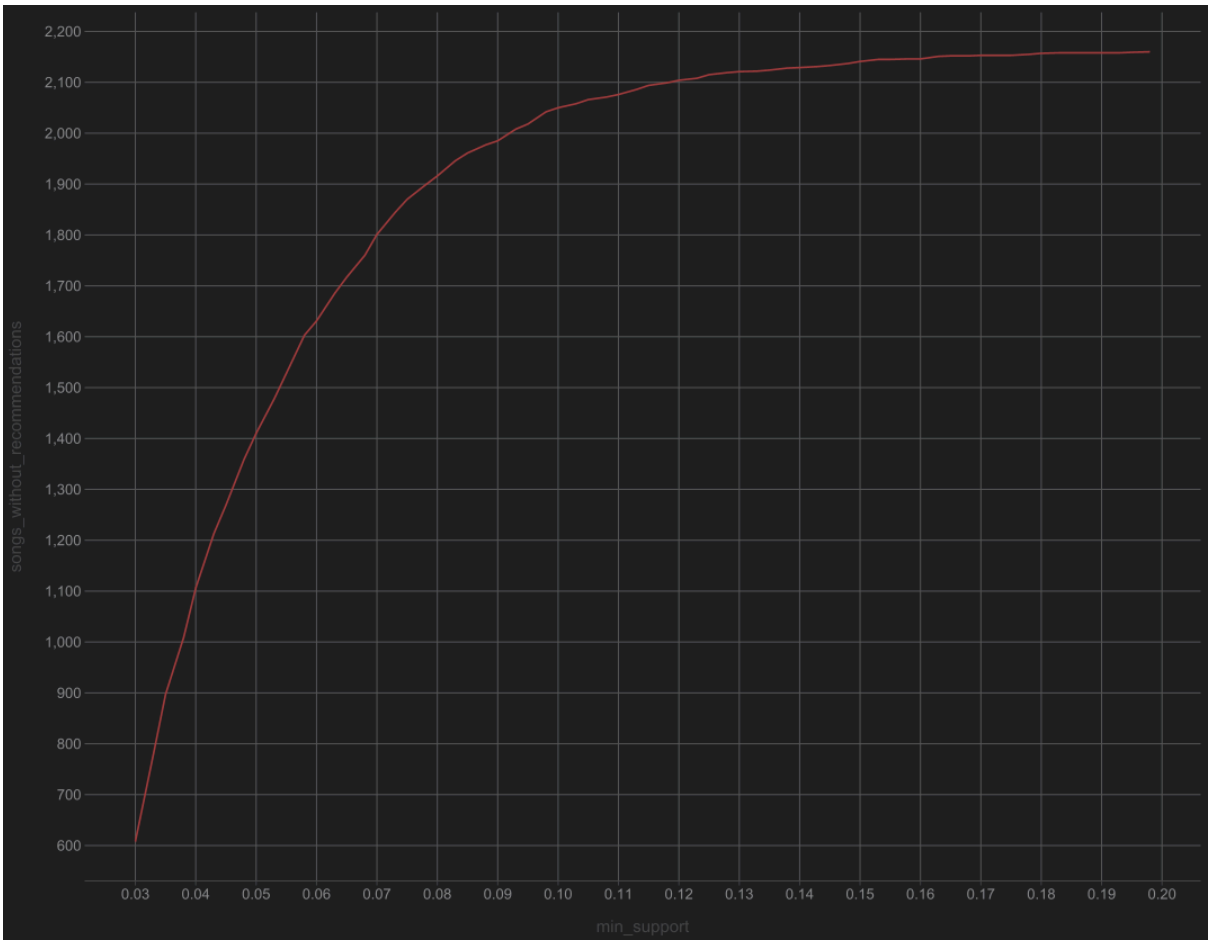


”

Otimização do algoritmo

O algoritmo de recomendação precisa ser preciso, gerando regras de recomendação para boa parte das músicas do dataset, e rápido, permitindo que os serviços consumidores atualizem seus dados assim que possível.

A biblioteca que utilizei trabalha com matrizes, tornando suas computações mais rápidas, bastando então estudar valores para o suporte mínimo.



Os valores de suporte são muito baixos, possivelmente pela baixa informatividade do modelo apenas com nomes, resultando em crescente perda de recomendações com o crescimento. Para conciliar tempo de execução e qualidade, escolhi 0.05.

Regeneração do modelo

O serviço de machine learning foi feito para ser executado como job, seja no ArgoCD ou agendado por algum cliente externo. Para fins de demonstração fiz ele se comportar como um cronjob, alterando o dataset a cada execução.

O caminho do dataset é passado como variável de ambiente e aponta para o PVC.

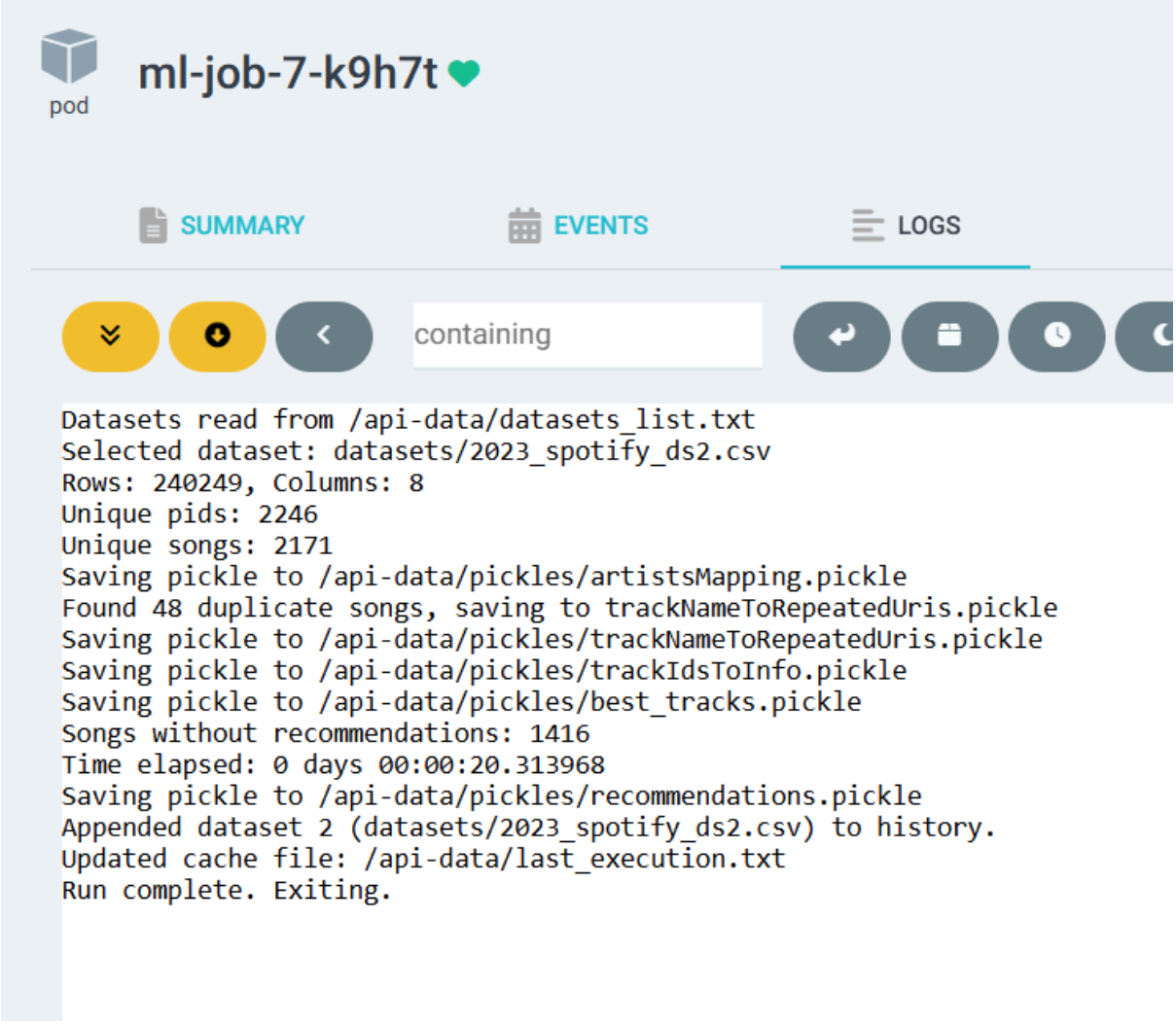
Um arquivo no PVC controla o histórico de execuções e outro contém a listagem de datasets. O último dataset do histórico é recuperado e o índice incrementado, de forma que as execuções alternam entre os datasets disponíveis.

A rotina de execução lê e prepara o dataset, roda o algoritmo fpgrowth e transforma as regras geradas para fácil computação posterior, salvando no volume. Ao longo da execução outros arquivos pickle intermediários são gravados para uso no frontend, se necessário.

Por fim, a execução é inserida no histórico e o job é encerrado. Foi configurado um TTL para removê-lo.

Com uma configuração de replace do ArgoCD, foi possível executar o serviço periodicamente. Isso foi atingido com o uso das configurações `argocd.argoproj.io/sync-options: Force=true,Replace=true` ao mesmo tempo que um TTL estava ativo, fazendo que sempre que o TTL fosse atingido o ArgoCD substituísse o job.

Logs para demonstração:



Exemplo de histórico de execuções

```
diogoneiss@cloud2:~/project2-pv2$ cat dataset_history.csv
time,dataset_index,dataset_file
2025-01-13 02:55:51,1,datasets/2023_spotify_ds1.csv
2025-01-13 03:06:21,2,datasets/2023_spotify_ds2.csv
2025-01-13 03:16:45,1,datasets/2023_spotify_ds1.csv
2025-01-13 03:27:13,2,datasets/2023_spotify_ds2.csv
2025-01-13 03:36:20,1,datasets/2023_spotify_ds1.csv
2025-01-13 03:46:49,2,datasets/2023_spotify_ds2.csv
2025-01-13 03:57:12,1,datasets/2023_spotify_ds1.csv
2025-01-13 04:07:40,2,datasets/2023_spotify_ds2.csv
```

Detecção de mudanças no backend

O caminho pensando inicialmente seria fazer um broadcast de requisições HTTP para todas as réplicas solicitando invalidação dos dados, porém a complexidade de fazer isso bem excedeu minhas habilidades com kubernetes

Optei para ir numa abordagem de polling, em que o serviço confere se os dados estão atualizados periodicamente de acordo com um arquivo gerado pelo serviço de machine learning.

A anotação `@repeat_every(seconds=60)` foi utilizada para implementar esse comportamento e testes foram feitos analisando os logs. Com o deploy feito, o valor fixo de 60 segundos foi substituído para uma variável de ambiente inicializada com 10 minutos.

Ao iniciar o servidor os dados mais recentes são sempre recuperados, de forma que os dados sejam eventualmente consistentes.

Logs no início

```
Mon, Jan 13 2025 9:21:07 am       INFO   Uvicorn running on http://0.0.0.0:80 (Press CTRL+C to quit)
Mon, Jan 13 2025 9:21:07 am  2025-01-13 12:21:07,941 - INFO - Data is stale, current value is None and last value was 2025-01-13 09:15:13
Mon, Jan 13 2025 9:21:07 am  2025-01-13 12:21:07,941 - INFO - Reloading data!
Mon, Jan 13 2025 9:21:07 am  2025-01-13 12:21:07,942 - INFO - Best tracks loaded: 2122                                    ^ leitura no startup começa cc
Mon, Jan 13 2025 9:21:07 am  2025-01-13 12:21:07,943 - INFO - Recommendations loaded: 755
Mon, Jan 13 2025 9:21:07 am  2025-01-13 12:21:07,943 - INFO - Finished reloading, should reflect changes. Data was reloaded 1 times
Mon, Jan 13 2025 9:26:07 am  2025-01-13 12:26:07,943 - INFO - Data is stale, current value is 2025-01-13 09:15:13 and last value was 2025-01-13 09:21:24
Mon, Jan 13 2025 9:26:07 am  2025-01-13 12:26:07,943 - INFO - Reloading data!
Mon, Jan 13 2025 9:26:07 am  2025-01-13 12:26:07,944 - INFO - Best tracks loaded: 2163
Mon, Jan 13 2025 9:26:07 am  2025-01-13 12:26:07,945 - INFO - Recommendations loaded: 753
Mon, Jan 13 2025 9:26:07 am  2025-01-13 12:26:07,945 - INFO - Finished reloading, should reflect changes. Data was reloaded 2 times
Mon, Jan 13 2025 9:31:07 am  2025-01-13 12:31:07,949 - INFO - data is not stale, no need to reload
Mon, Jan 13 2025 9:36:07 am  2025-01-13 12:36:07,950 - INFO - data is not stale, no need to reload
Mon, Jan 13 2025 9:41:07 am  2025-01-13 12:41:07,952 - INFO - data is not stale, no need to reload
Mon, Jan 13 2025 9:46:07 am  2025-01-13 12:46:07,954 - INFO - Data is stale, current value is 2025-01-13 09:21:24 and last value was 2025-01-13 09:41:54
Mon, Jan 13 2025 9:46:07 am  2025-01-13 12:46:07,954 - INFO - Reloading data!
Mon, Jan 13 2025 9:46:07 am  2025-01-13 12:46:07,955 - INFO - Best tracks loaded: 2122
```

Logs depois de um tempo

```
Mon, Jan 13 2025 10:36:07 am  2025-01-13 13:36:07,980 - INFO - data is not stale, no need to reload
Mon, Jan 13 2025 10:41:07 am  2025-01-13 13:41:07,981 - INFO - data is not stale, no need to reload
Mon, Jan 13 2025 10:46:07 am  2025-01-13 13:46:07,982 - INFO - Data is stale, current value is 2025-01-13 10:22:46 and last value was 2025-01-13 10:43:09
Mon, Jan 13 2025 10:46:07 am  2025-01-13 13:46:07,982 - INFO - Reloading data!
Mon, Jan 13 2025 10:46:07 am  2025-01-13 13:46:07,982 - INFO - Best tracks loaded: 2163
Mon, Jan 13 2025 10:46:07 am  2025-01-13 13:46:07,983 - INFO - Recommendations loaded: 753
Mon, Jan 13 2025 10:46:07 am  2025-01-13 13:46:07,983 - INFO - Finished reloading, should reflect changes. Data was reloaded 6 times
```

O próprio serviço decide se os dados são stale ou não com base no arquivo de controle, como pode ser visto nessa amostra dos logs.

Riscos da abordagem incluem

1. Problemas de concorrência, em que clientes são afetados por leitura suja ou alguma inconsistência de dados ocasionada pelo disparo da rotina de alteração de dados. Isso poderia ser resolvido com locks, porém essa implementação foge do escopo do trabalho.
2. Regras não inicializadas, o que pode ocorrer se forem executados a partir de um cluster recém inicializado, em que o PVC ainda não contém as regras. Caso isso aconteça hoje, o serviço ficará em loop de crashes até o job finalizar a execução, o que é rápido. Uma solução em potencial seria automaticamente copiar um conjunto de regras cru para o PVC caso ele esteja vazio, porém isso fugiu do escopo do trabalho.