DCC831 TECC: Computação em Nuvem

# TP2 - DevOps and Cloud Computing

Gabriel de Oliveira Andrade

Aluno: Gabriel de Oliveira Andrade

Professor: Italo Fernando Scota Cunha

BELO HORIZONTE

Univerisidade Federal de Minas Gerais - 2025/1

# Lista de Figuras

# Sumário

# Introdução

Neste trabalho prático de DevOps e Cloud Computing, foi desenvolvido um sistema simples de recomendação de músicas utilizando contêineres Docker, Kubernetes (K8s) e ArgoCD. O principal objetivo foi compreender o funcionamento de um pipeline automatizado de implantação contínua (CI/CD) e como essas tecnologias se integram para manter uma aplicação sempre disponível, atualizada e escalável dentro de um cluster Kubernetes. Além disso, foi possível praticar conceitos como versionamento, observação de downtime e atualização automática de imagens a partir de commits no repositório.

O sistema completo é composto por três principais componentes: o frontend container, o backend container e a camada de orquestração no Kubernetes. O frontend recebe as requisições HTTP, processa as músicas enviadas e retorna as recomendações através do uso do arquivo *rules.pickle*, que é gerado no backend do sistema. Já o backend processa os dados de playlists de músicas disponibilizadas e através do treinamento de Itemset Mining, que gera o arquivo utilizado pelo Frontend para gerar as recomendações. O ArgoCD é responsável por monitorar o repositório no GitHub e atualizar automaticamente o cluster quando há uma mudança relevante no repositório, o que mantém a aplicação em execução sincronizada com o código mais recente.

O modelo de recomendação foi implementado no arquivo $frontend_container/app.py$. Nele há uma lógica que cria recomendações a partir de uma lista de músicas enviadas pelo usuário via requisição POST. O código percorre regras de associação e verifica se determinadas músicas enviadas estão contidas no conjunto de regras criado pelo backend. Caso a música esteja presente, ele retorna as próximas músicas mais prováveis, ordenadas por confiança. O resultado retornado é uma lista das músicas recomendadas, junto com a versão atual da aplicação e a data do modelo em formato JSON. Existe também a opção de se retornar mais ou menos músicas. O número padrão de músicas retornadas por playlist é 3.

O código pode ser observado nesse repositório.

Durante o processo, foram realizados testes de atualização contínua para observar o tempo de resposta e identificar eventuais períodos de indisponibilidade da aplicação.

# Discussão

## Discussion of test cases performed with Kubernetes and ArcoCD to exercise continuous integration functionalities and expected results.

Foram realizadas medidas mostradas na seção abaixo . A funcionalidade do ArgoCD também foi testada por mim durante o desenvolvimento desse trabalho e a opção de Auto Prune se mostrou particularmente útil para o contexto do trabalho realizado, pois antes do seu uso, versões paralelas de deployments eram mantidas ao mesmo tempo. Conforme mostrado na seção abaixo, houve um momento somente no qual os resultados esperados não foram obtidos (observado e explicado na seção de update da versão), e infelizmente, por motivo de tempo, não será possível a realização do teste em tempo hábil para adicionar

nesse relatório.

Achei a estratégia de ArgoCD muito interessante também por realizar a integração de CI/CD muito facilitada, algo que eu pessoalmente não havia trabalhado até então.

## How long it takes for changes to your code to be deployed and whether the application becomes offline. Please report your findings when you (i) update the version of the code (model), (ii) update the deployment (e.g., number of Kubernetes replicas), and (iii) update the dataset used by the model.

Nessa seção foram criados scripts para que a medida se tornasse eficiente. Os scripts têm o intuito de fazer o commit e push no github e logo após começarem a monitorar por mudanças no cluster, tudo isso enquanto um timer é ativado. Assim que as mudanças são finalizadas, o timer é parado e o tempo final é obtido. O script também menciona qualquer período de downtime que possa existir.

O arquivo $measurement_test.bash$ foi criado para medir o tempo que o sistema leva para atualizar a versão da aplicação durante o processo de deploy automático com o ArgoCD.

As mudanças nos arquivos do *deployment.yaml* são feitas manualmente e o commit é feito no script, que então faz push das mudanças para o GitHub. Depois disso, chamadas do meu computador pessoal são realizadas no cluster. Isso é possível através do comando, que replica o endereço do serviço dos Pods para o localhost da minha máquina, me permitindo fazer requests localmente.

```
ssh -fNT -L 50010:10.43.155.95:50010 gabrielandrade@cloudvm2
```

O script então faz requisições repetidas para o endpoint da API (com curl) e verifica a versão retornada pelo servidor. Ele começa marcando o tempo inicial e, a cada requisição, imprime o tempo decorrido e a versão atual recebida.

O loop continua até que a versão da aplicação mude em relação à original (por exemplo, de 1.0.0 para uma nova versão). Quando isso acontece, o script exibe o tempo total gasto para a atualização. Caso o servidor fique fora do ar durante o processo, o script também detecta o downtime e mostra na tela o tempo em que o sistema ficou indisponível.

Dessa forma é possível observar o comportamento do pipeline CI/CD — mostrando quanto tempo o sistema leva para aplicar uma nova versão e se há interrupções no serviço durante a atualização.

A mudança manual para cada uma das situações mencionadas no enunciado foi feita de maneira diferente, o que permite mensurarmos o tempo que cada uma delas leva para o resultado final. O script mencionado acima foi usado para a atualização de código e atualização de dataset. Já para a atualização do deployment, como o número de réplicas, foi criado um script separado.

Já o script que detecta o número de réplicas alteradas, a lógica utilizada foi baseada na saída do comando a seguir, que retorna exatamente o número de réplicas existentes.

```
kubectl -n gabrielandrade get deploy
```

```
song-recommender-goa-deployment-ds1-v1.0.2  -o
jsonpath='{.status.availableReplicas}'
```

—

O restante da lógica foi bem parecido, de forma que um loop havia sido feito, com atualizações a cada segundo, relatando se a mudança já havia ocorrido ou não. A lógica de mudança de número de réplicas foi pensada dessa forma, pois a versão ainda poderia ser a mesma, o que inviabilizaria o uso do script acima.

A principal diferença é que o script de medidas fica no servidor, enquanto o commit e push são feitos na minha máquina local, e logo após o script, no servidor, é iniciado, através do comando a seguir, que foi adicionado no script bash. Ele inicia remotamente o script no servidor e printa a saída na minha máquina local, me permitindo ver o tempo passado.

```
ssh gabrielandrade@cloudvm2 "bash ~/measure_scale.sh"
```

Para alguns scripts, foram necessárias mais de uma tentativa, o que foi adicionado na mensagem de commit como - att X". O interessante dessa abordagem é que é possível também acompanhar as mudanças pelo repositório do Github, que mantém o histórico dos commits e suas mensagens.

### Update the version of the code (model)

Como a versão do código é obtida através de uma variável de ambiente no arquivo de Deployment, temos que a operação de atualizar a versão do código vem juntamente com a mudança de nome do Deployment, conforme foi recomendado no arquivo de descrição do TP2. Dessa forma, o script abaixo monitora também a existência de downtime no período e quanto tempo levaria até que a nova versão fosse implantada.

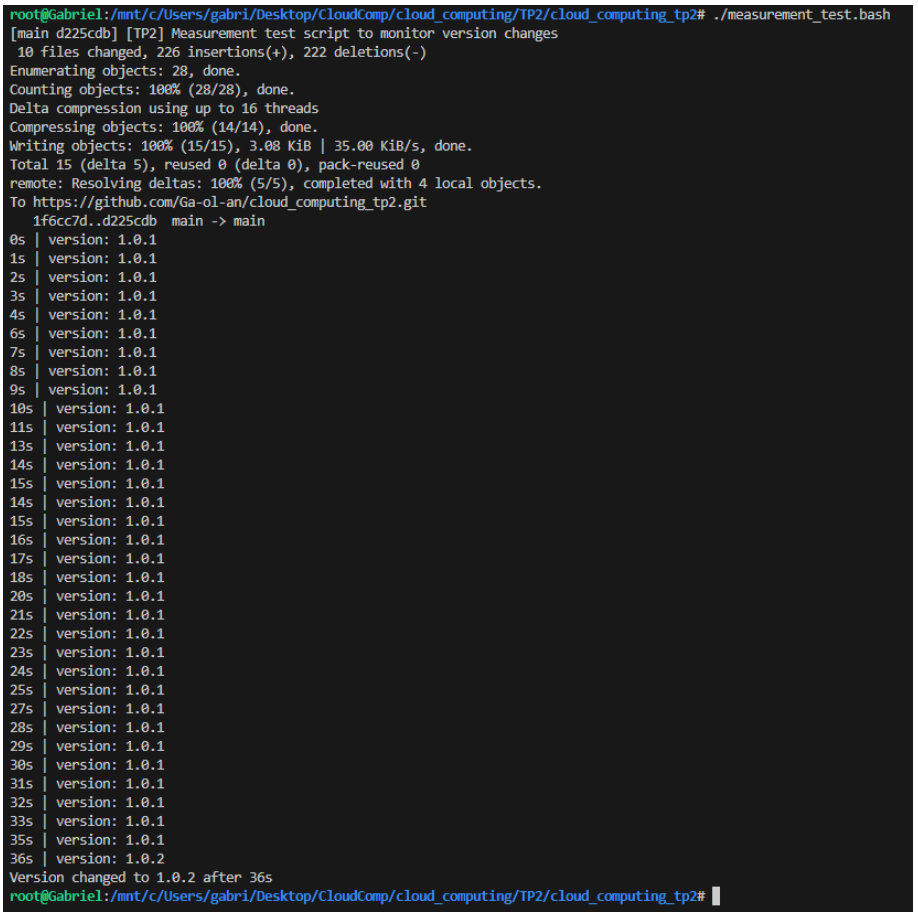


Figura 1: Saída do script que realiza as medições para o caso da mudança de versão

Conforme podemos observar na Fig. 1, não houve nenhum downtime durante o período e toda a mudança aconteceu em 36 segundos. A não existência de um downtime não era esperada, pois, como havia somente 1 Pod, em teoria, esse Pod deveria ser finalizado para que o novo fosse adicionado. Acredito que isso possa ser atribuído ao tempo de verificação, de 1 segundo, e o Pod pode ter sido trocado em algum período de tempo inferior a esse, dado que todas as demais características do POD eram as mesmas. Por falta de tempo, não será possível refazer o teste para adicionar os resultados nesse relatório. O seguinte seria feito: O parâmetro sleep seria mudado de 1 para 0.1, de forma que haveria uma maior granularidade nos testes, permitindo que o momento exato de downtime do Pod fosse notado.

### Update the deployment

Para o teste de atualização do deployment, foi alterado o número de réplicas de 1 para 2, e o teste realizado visava a medir quanto tempo levaria até que tivéssemos duas réplicas do deployment presentes no cluster.

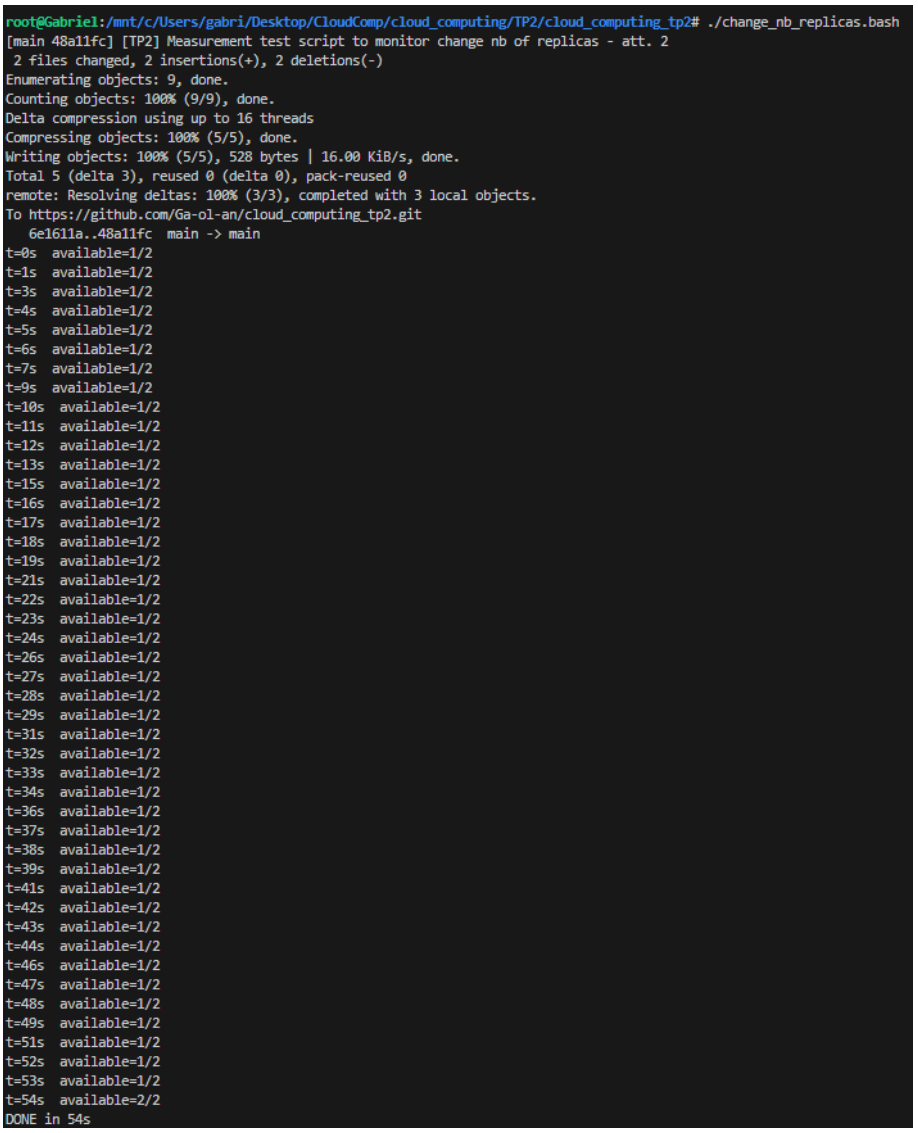


Figura 2: Saída do script que realiza as medições para o caso da mudança do deployment

Conforme podemos ver na Figura 2, todas as mudanças levaram 54 segundos para ocorrerem.

### Update the dataset used by the model

A atualização do dataset foi feita através da mudança da variável $DATASET\_PATH$, no arquivo de deployment. Como explicado no arquivo de descrição do TP2, como variáveis ambiente não fazem com que o Pod seja também recriado, foi alterado também o nome do deployment, de forma que o Pod seja recriado para a nova versão.

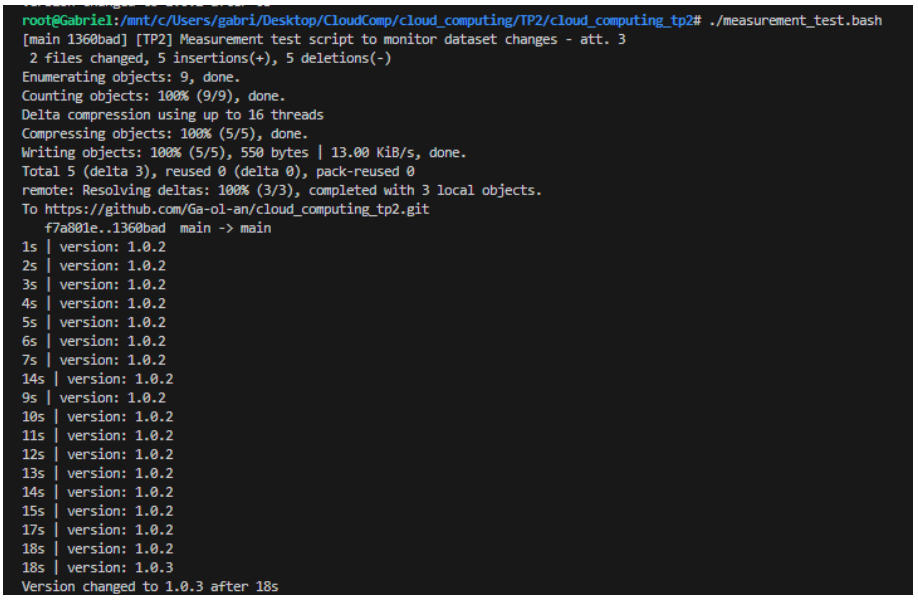


Figura 3: Saída do script que realiza as medições para o caso da mudança do dataset

O tempo notado para que toda a operação fosse realizada foi de 18 segundos. Exatamente metade do tempo notado para a seção de update da versão do deployment. Algo a ser mencionado é que, como ambas as atualizações requeriam a recriação do deployment, o tempo esperado para ambas deveria ser o mesmo. O que pode explicar tal diferença no tempo é que, para o primeiro teste, realizado na seção de update de versão, havia somente 1 réplica rodando. Já para o teste de Dataset, havia 2 réplicas rodando. Dessa forma, com apenas uma réplica, o ciclo de atualização ocorre de forma sequencial dado que somente um Pod existe e o tempo de downtime tenta ser minimizado ou zerado, o que faz com que o tempo total de implantação demore mais.

Por outro lado, quando o deployment tem duas réplicas, o Kubernetes consegue realizar a atualização de forma paralela, mantendo uma instância antiga ativa enquanto a nova é criada e preparada. Esse comportamento reduz o tempo total percebido, já que parte das operações ocorre simultaneamente.

## How you detect model changes in the REST API back-end.

O código do frontend pega a versão através de uma variável ambiente presente no *deployment.yaml*. Depois disso, essa versão é retornada no campo *version*, no Json retornado na saída do código. Já o código de backend não detecta explicitamente a mudança de versão, mas conforme configurado no ArgoCD, caso haja uma mudança de versão, o deployment deve ser recriado, o que faria o backend se atualizar conforme necessário.

### How your containers obtain the new dataset when regenerating the model.

Existe uma variável ambiente no arquivo *deployment.yaml* que dita o endereço de URL do dataset a ser utilizado. Dessa forma, quando o código de backend for iniciado, ele usará essa variável para definir qual dataset baixar e realizar os treinamentos.

## Conclusão

A realização deste trabalho permitiu aprofundar meus conhecimentos em Kubernetes e ArgoCD, compreendendo na prática como essas ferramentas se conectam para automatizar e gerenciar implantações. O tema proposto, voltado à criação de um modelo de recomendação de músicas, foi interessante e ajudou a tornar o estudo mais envolvente. Essa combinação entre infraestrutura e lógica de aplicação tornou o aprendizado mais concreto, mostrando de forma clara o impacto das práticas de integração e entrega contínua no funcionamento de sistemas modernos. Foi também interessante monitorar o funcionamento do CICD e como o Kubernetes se atualiza de maneira otimizada e inteligente.

# Referências

[1]Kubernetes Docs – Rolling Updates