Project 2: DevOps and Cloud Computing: Pipeline de DevOps e MLOps para Recomendação de Playlists
DCC - UFMG
Belo Horizonte

Lucas Albuquerque Santos Costa (lucascosta)
2023028005

1 Introdução

Este documento descreve o *pipeline* de Integração Contínua e Entrega Contínua (CI/CD) construído para o Projeto 2. O objetivo foi desenhar, implementar e implantar um serviço de recomendação de *playlists* baseado em microsserviços (API e Treinamento de ML) no Kubernetes, utilizando o ArgoCD como ferramenta de GitOps.

O *pipeline* final é totalmente automatizado: mudanças no repositório Git (seja no código da aplicação, seja na configuração do Kubernetes) são detetadas pelo ArgoCD e aplicadas ao *cluster* automaticamente.

2 Discussão dos Testes de CI/CD

Para validar o *pipeline* de MLOps, foram realizados os três testes de integração e entrega contínua, conforme solicitado no enunciado.

2.1 Teste 1: Atualização da Configuração (Réplicas)

O primeiro teste consistiu em verificar se o ArgoCD detetava e aplicava mudanças na configuração do Kubernetes.

- **Ação:** O arquivo `deployment.yaml` no repositório Git foi modificado, alterando o número de réplicas da API (de 1 para 2, e posteriormente de 2 para 1).
- **Resultado Esperado:** O ArgoCD deveria sincronizar a mudança e o Kubernetes deveria ajustar o número de *pods* da API em conformidade.
- **Resultado Obtido:** Ação executada com sucesso. Após o `git push` e a sincronização do ArgoCD, o comando `kubectl get pods` confirmou que o número de *pods* da API foi ajustado (aumentado para 2 e depois diminuído para 1), provando que o *pipeline* de GitOps para configuração do *cluster* estava funcional.

2.2 Teste 2: Atualização do Código (Imagem Docker)

O segundo teste visou validar o *pipeline* de atualização da versão do código da aplicação (a API).

- **Ação:** Novas imagens Docker da API (ex: `...:0.6`, `...:0.7`) foram construídas e enviadas ao Docker Hub. O arquivo `deployment.yaml` no Git foi atualizado para usar a nova *tag* de imagem e também para refletir a nova versão na variável de ambiente `CODE_VERSION`.
- **Resultado Esperado:** O ArgoCD deveria detetar a mudança no *deployment* e o Kubernetes deveria realizar um *rolling update*, substituindo os *pods* antigos pelos novos sem interromper o serviço.
- **Resultado Obtido:** Sucesso. O ArgoCD sincronizou a mudança. O Kubernetes iniciou os novos *pods* (com a nova imagem) e, somente após eles estarem `Running` e saudáveis, ele começou a terminar (`Terminating`) os *pods* antigos. O teste com `wget` confirmou que a API estava a servir a nova versão (ex: `"version":"0.7.0"`).

2.3 Teste 3: Atualização do Dataset (Modelo de ML)

O terceiro teste validou o *pipeline* de MLOps: a atualização do modelo de *Machine Learning* através da troca do *dataset*.

- **Ação:** No repositório Git, o arquivo `job-ml-ds1.yaml` foi removido e substituído pelo `job-ml-ds2.yaml`. O *script* `train.py` foi modificado para treinar numa **amostra aleatória** do *dataset* (definida pela variável `MAX_BASKETS: 30000`), permitindo o uso de um `MIN_SUP` baixo (0.5%) dentro do limite de memória de 2G.
- **Resultado Esperado:** O ArgoCD (configurado com `auto-prune`) deveria apagar o Job antigo (DS1) e criar o novo Job (DS2). O novo Job (com a lógica de amostragem) deveria treinar o modelo com o *dataset* DS2, e a API (que estava `Running`) deveria detetar a mudança e carregar o novo modelo.
- **Resultado Obtido:** **Sucesso total.** O ArgoCD executou a troca dos Jobs. O Job DS2 (com a lógica de amostragem) rodou e foi concluído (`Completed`). O teste final com `wget` (usando "Yesterday"e "Bohemian Rhapsody"como entrada) retornou um JSON completo, provando que o modelo estava a funcionar e a gerar recomendações. A resposta obtida foi:

  `"debug_info":"Tested 2 songs against 2119 rule antecedents", ...`

  `"model_date":"2025-11-09T00:22:03Z", ...`

  `"songs":["sweet home alabama", "sweet child o' mine", ...], ...`

  `"total_rules": 2119, "version":"0.4.0"`

  Isto confirmou que 2119 regras foram geradas e que a API estava a servir recomendações reais.

3 Tempo de Deploy e Downtime

A monitorização do *pipeline* durante os testes revelou os seguintes dados:

- **Tempo de Sincronização (ArgoCD):** A sincronização do ArgoCD (após o `git push`) foi quase instantânea, levando entre 0 e 2 segundos para aplicar as mudanças no *cluster*.
- **Tempo de Atualização (Kubernetes):**
  - **Réplicas (Teste 1):** O *deploy* de um novo *pod* da API foi muito rápido, levando cerca de 10-15 segundos para o novo *pod* estar `Running`.
  - **Imagem (Teste 2):** O *rolling update* completo (iniciar novos *pods* e terminar os antigos) levou cerca de 30-45 segundos.
  - **Modelo (Teste 3):** Esta foi a atualização mais longa, dominada pelo tempo de execução do Job de ML (cerca de 2-3 minutos para o *pod* do Job ir de `Pending` para `Completed`).
- **Downtime (Interrupção do Serviço):** Não houve **nenhum downtime** mensurável em nenhuma das três atualizações.
  - Nos Testes 1 e 2, o mecanismo de *rolling update* do Kubernetes garantiu que o tráfego só fosse enviado para os novos *pods* quando eles estavam prontos, e os antigos só eram desligados depois disso.

– No Teste 3, a API (um `Deployment`) permaneceu `Running` o tempo todo, enquanto o treino (um `Job`) ocorria em segundo plano. A API só recarregou o modelo após o novo arquivo estar completo, garantindo uma transição suave.

4 Deteção de Mudanças no Modelo (API)

O enunciado exigia que a API recarregasse o modelo de ML sempre que ele mudasse, sem a necessidade de reiniciar o *pod*.

Esta funcionalidade foi implementada no `api/app.py` usando um *thread* de *background* (do módulo `threading` do Python), iniciado com a API.

1. A função `_watch_model` corre num *loop* infinito (configurado como *daemon*).
2. A cada 5 segundos, ela verifica o *timestamp* de modificação do arquivo (`os.path.getmtime`) em `/shared/model/rules_model.pkl`.
3. Se o *timestamp* do arquivo for diferente do *timestamp* do modelo em memória, a API recarrega o modelo usando `pickle.load` e atualiza o *timestamp* interno.

O sucesso do Teste 3 (onde o `model_date` mudou na resposta do `wget` sem que a API reiniciasse) provou que este mecanismo de *watcher* funciona perfeitamente.

5 Obtenção do Dataset pelo Contêiner de ML

O contêiner de ML (imagem `lukasalbukk/playlist-ml`) não armazena o *dataset* internamente. O *pipeline* foi desenhado para que o *dataset* seja fornecido em tempo de execução.

No *pipeline* final, o método utilizado foi o download direto.

1. O arquivo YAML do Job (ex: `job-ml-ds2.yaml`) define uma variável de ambiente chamada `DATASET_URL`.
2. O valor dessa variável é o link "cru"(raw) para o arquivo `.csv` hospedado diretamente no repositório GitHub (ex: `https://raw.github.../ds2.csv`).
3. O *script* `train.py` lê esta variável (`os.getenv`) e, na função `_download_or_open`, usa a biblioteca `requests` do Python para baixar o arquivo pela internet antes de o carregar no Pandas.

Esta abordagem é superior a usar um volume, pois permite que o *dataset* seja atualizado no Git sem exigir que os arquivos sejam copiados manualmente para o volume do *cluster*.