Practical Work 2 in Cloud Computing

Gustavo Freitas Cunha

Federal University of Minas Gerais (UFMG)
Belo Horizonte - MG - Brazil

gustavocunha@dcc.ufmg.br

1 Accessing the Cluster and Management Tools

In this section, I will describe how I accessed the Kubernetes cluster and the management tools (ArgoCD and Rancher) as part of the practical work (TP2). I will explain the steps I took to establish SSH access, as well as how I used ArgoCD and Rancher for deployment management.

1.1 Accessing the Cluster via SSH

To access the cluster, I first needed to establish an SSH connection with the virtual machine (VM) where the Kubernetes cluster is hosted.

I made sure to keep port mappings in mind, especially since the Kubernetes services are mapped to specific ports, such as `31443` for ArgoCD and `30443` for Rancher.

1.2 Accessing ArgoCD

After establishing the SSH connection, I accessed ArgoCD by opening a web browser and connecting to `localhost:31443`. Once I logged in, I was taken to the ArgoCD dashboard. The dashboard initially appeared empty, as no applications had been deployed yet. I then clicked on the "Create Application" button to start configuring my application deployment. After configuring the application, I was able to deploy it and monitor its status directly from the ArgoCD interface.

1.3 Accessing Rancher

To access Rancher, I opened the browser and connected to `localhost:30443`, the port used by Rancher, following the same setup of the SSH tunnel. After setting the new password, I was taken to the Rancher dashboard.

From there, I navigated to the "Cluster" section and selected the local cluster to view the available dashboards. I went to "Workloads" and then "Pods" to check the status of the pods I had deployed via ArgoCD.

This integration of ArgoCD and Rancher made it easy to monitor the status and logs of my applications running on the Kubernetes cluster and greatly simplified the process of managing deployments. By the way, I discovered three mistakes that I have made without realizing what I was doing when I was developing locally.

2 Detecting Model and Dataset Changes

2.1 Detecting Model Changes in the REST API Back-End

In the REST API back-end, model changes are detected by checking for modifications to the model file stored in the PVC. When the API starts, the model file, located at `/app/data/rules.pkl`, is loaded, and its content is used to generate playlist recommendations. To detect changes, the API calculates the checksum of the model file and stores it in memory. On subsequent API requests, the checksum of the current model file is recalculated and compared with the stored checksum. If the checksums differ, this indicates that the model has been updated, prompting the API to reload the new version of the model into memory. The model's checksum logic is implemented in the wsgi.py file of the API.

2.2 Handling Dataset Updates

The new dataset for the model is obtained and processed through a machine learning job that runs whenever there is an update to the dataset. This job is managed by a Kubernetes Job resource, specified in the file `ml-job.yaml`, which handles the execution of the model training process. The dataset is periodically (every five minutes) checked for updates by a Kubernetes CronJob, defined in the file `ml-job-trigger.yaml`. The CronJob downloads the latest version of the dataset from a specified URL and compares its checksum with the previously stored value. If the dataset has changed, the CronJob triggers the job defined in `ml-job.yaml` to retrain the model. The ML container, using the image `gustavofcunha/recommendation_rules:v3`, is responsible for regenerating the model. The job mounts the shared PVC at `/modelo/data/rules.pkl` to save the newly generated model. After the new model is saved, the REST API back-end detects the change by comparing the model's checksum to the previously stored checksum. Once a change is detected, the API reloads the new model and starts using it for future recommendation requests.

3 Kubernetes and ArgoCD Test Cases

During testing, I exercised the continuous integration functionalities of Kubernetes and ArgoCD by performing several updates and observing the system's response. The following test cases were conducted to evaluate how changes to the code (model), deployment, and dataset are handled, and their impact on the application's availability.

3.1 Updating the Version of the Code (Model)

Updates to the model's code are managed through ArgoCD, which continuously monitors the repository for changes. When a new commit is pushed to the repository containing changes to the model's code, ArgoCD detects the update and initiates a new deployment. The deployment process, which involves replacing the running pods with new ones based on the updated code, took approximately 3-5 minutes. During this time, the application remained online, as the rolling update strategy ensured that older pods continued serving requests until the new ones were fully operational. This strategy minimized disruption and maintained the application's availability.

3.2 Updating the Deployment Configuration

Changes to the deployment configuration, such as modifying the number of Kubernetes replicas, were tested by updating the deployment file and applying the changes using `kubectl`. Kubernetes

began updating the replicas immediately, scaling the pods up or down to match the desired configuration. This process took approximately 2-4 minutes and was also seamless, with no observed downtime. The rolling update mechanism allowed the application to remain responsive throughout the scaling process.

3.3 Updating the Dataset Used by the Model

Dataset updates are handled by a Kubernetes CronJob defined in `ml-job-trigger.yaml`, which periodically (every five minutes) checks for new versions of the dataset. When an updated dataset is detected, the CronJob triggers the execution of a Kubernetes Job defined in `ml-job.yaml`. This job runs the machine learning container (`gustavofcunha/recommendation_rules:v3`) to retrain the model using the new dataset. The retraining process typically took around 10-15 minutes. During this time, the REST API server continued serving requests using the previously trained model, ensuring uninterrupted service. Once the retraining process was completed, the newly generated model file (`rules.pkl`) was saved to the shared PVC.

The REST API back-end detected updates to the model by utilizing a custom function defined in the `wsgi.py` file. This function calculates the checksum of the model file, `rules.pkl`, stored in the PVC mounted at `/app/data/rules.pkl`. The checksum is generated using the `hashlib.md5` library, ensuring a unique hash for each version of the model.

At the application startup, the model is loaded into memory along with its associated metadata, including version and date. The checksum of the file is also computed and stored in memory as `last_checksum`. For every incoming request, the function compares the current checksum of the file against the stored `last_checksum`. If a discrepancy is detected, indicating that the model has been updated, the function reloads the model into memory by invoking the `load_model()` function, which reads the model file, unpacks the rules and metadata, and updates the in-memory state.

This mechanism ensures that the API dynamically adapts to model updates without requiring a restart, maintaining high availability while serving recommendations based on the latest model.

3.4 Summary of Findings

- **Code Updates:** Managed by ArgoCD. Changes in the repository triggered a deployment that completed in 3-5 minutes without downtime, utilizing a rolling update strategy.
- **Deployment Updates:** Handled via Kubernetes. Adjustments to replica counts were applied within 2-4 minutes seamlessly, with no interruption to the service.
- **Dataset Updates:** Managed by a Kubernetes CronJob and Job. Retraining the model took 10-15 minutes, with the API continuing to serve requests during the process. The updated model was detected and reloaded automatically.