Project 2: DevOps and Cloud Computing

Introduction

DevOps is a combination of practices and tools that increases an organization's ability to deliver applications and services at high velocity: helping organizations evolve and improve products at a faster pace than other organizations using traditional software development and infrastructure management processes.

Cloud computing is an important technology that aids in just about every step of a successful DevOps operation. Cloud computing enables collaboration without all the downtime of sending files back and forth to team members. The cloud, along with other tools like version control (Git), containers (Docker and Kubernetes), and patterns like microservices allows for simultaneous development and advanced experimental test environments for quickly prototyping solutions, enabling frequent updates, and speeding up delivery. Figure 1 exemplifies the activities in Continuous Integration (CI) and Continuous Delivery (CD), the two core concepts of the DevOps methodology.

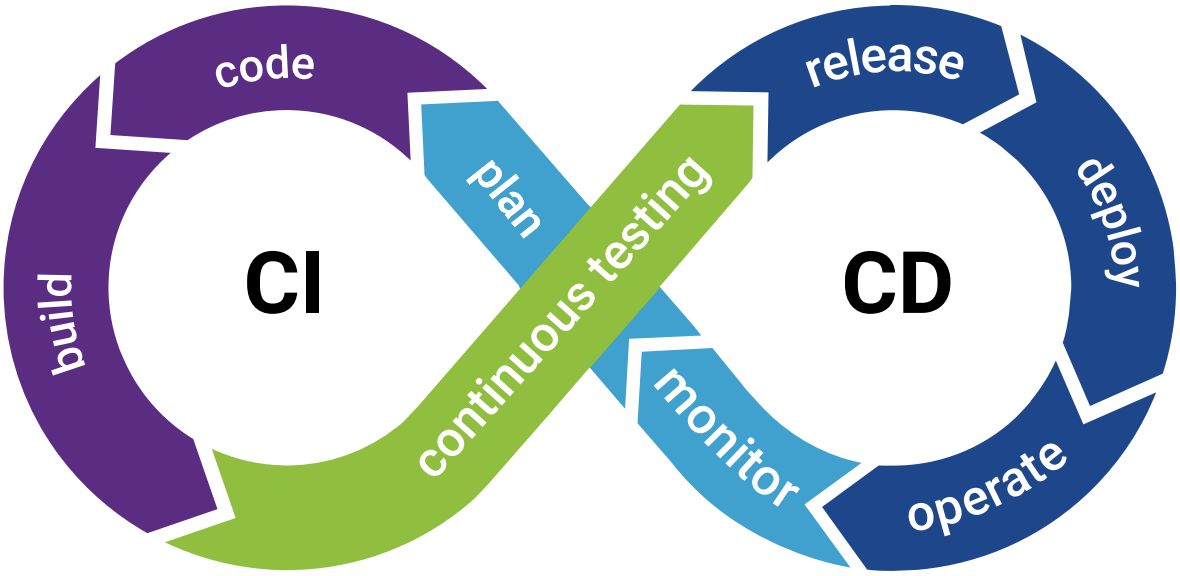

Figure 1. DevOps Pipeline.

Continuous integration is a practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. The key goals of continuous integration are to find and address bugs quicker, improve software quality, and reduce the time it takes to validate and release new software updates. *Continuous delivery* complements of continuous integration. Continuous delivery consists of automating deployment actions that were previously performed manually.

General Goal

In this project, students will design, implement and deploy a playlist recommendation service built on microservices, combining a Web front end and a machine learning module. The service will be built and tested using continuous integration, and automatically deployed using continuous delivery. The practice of integrating a machine learning workflow with DevOps has been referred to as MLOps.

You will obtain experience using some of the most popular tools in this context: Docker to containerize application components, Kubernetes to orchestrate the deployment in a cloud environment, GitHub (or another Git provider) as a central code repository, and ArgoCD as the continuous delivery framework on top of Kubernetes.

Students will create a recommendation system to recommend playlists to a user based on a set of songs that the user listened in the past.

Dataset

We will use a (small) sample of the Spotify dataset used in Project 1. The dataset sample is available on the cluster at `/home/datasets/spotify` (in the VM's root file system, not HDFS). It includes two parts:

1. The `2023_spotify_ds1.csv` and `2023_spotify_ds2.csv` datasets contain about 240000 playlists each. The two datasets represent the set of playlists in the platform and can be used to emulate an update to the model.
2. The `2023_spotify_songs.csv` file contains about 7000 songs in the playlists that can be used in the recommendation step.

> You can first train your model using `2023_spotify_ds1.csv` and later update the model using `2023_spotify_ds2.csv` .

The dataset is also hosted online at this location.

Part 1: Software Components

We will implement our playlist recommendation service as a microservice-based application. The application will have three components described next. These software components are mostly independent, so the microservice approach is a good fit.

1. Playlist Rules Generator

We will develop code to generate association rules for the recommendation system to use. More specifically, we will apply a Frequent Itemset Mining algorithm to find frequent patterns and their rules (if A, then B), which will ultimately allow the recommendation of playlists that users may enjoy.

> Frequent Itemset Mining (FIM) is the task of extracting any existing frequent itemset (a set of items having an occurrence frequency no less than some threshold) in data. It is a common task within data analysis since it is responsible for extracting frequently occurring events, patterns or items. Insights from such pattern analysis offer important benefits in decision-making processes. The generated pattern can be used to derive association rules to find interesting associations. For instance, consider the bellow example of five baskets of items:
>
> | ID | Items |
> |---|---|
> | 1 | Bread, Milk |
> | 2 | Bread, Diaper, Beer, Eggs |
> | 3 | Milk, Diaper, Beer, Coke |
> | 4 | Bread, Milk, Diaper, Beer |
> | 5 | Bread, Diaper, Milk, Coke |
>
> Considering this example, we can say that a frequency of co-occurrence of items `{Milk, Bread, Diaper}` is 2 (because all items occurred together in IDs 4 and 5). Its support (defined as how frequently a collection of items occurs together as a percentage of all transactions) is 2/5 = 0.4. If we define an itemset is frequent if they have a minimum frequency of 40% (i.e., a minimum support), the collection `{Milk, Bread, Diaper}` will be considered a frequent itemset.
>
> Moreover, from this collection we can generate possible association rules in the format `if (antecedent) then (consequent)`, like: `if ({Milk, Diaper}) then (Beer)`, `if (Milk) then ({Beer, Diaper})`, `if (Diaper) then (Beer)`, or any other combination between their items. However, not all rules are good matches for the data. A popular metric to evaluate an association rule is *confidence*, defined as the ratio of the number of transactions that includes all items in `consequent` and `antecedent` (i.e., **`consequent`** ∪ **`antecedent`**) by the number of transactions that includes all items in `consequent`. For instance, the confidence of rule `if ({Milk, Diaper}) then (Beer)` is 2/3 = 0.67, in other words, beer was purchased 67% of the time when Milk and Diaper were bought.
>
> You are free choose the best value for the support threshold, the confidence, or to improve your recommendation system (e.g., by combining multiple or using more advanced machine learning approaches).

There are different Frequent Itemset Mining algorithms such as, Apriori, Eclat and FP-Growth. Students are free to use whatever algorithm and ML framework they want. Two popular solutions in Python are the Apriori algorithm in `mlxtend` and the FP-Growth algorithm in `fpgrowth-py`.

> As an example, if using the fpgrowth-py module, we can represent a set of baskets (i.e., each row represents a basket and contains the items in the basket) and get the frequent itemsets as below:
>
> ```
> from fpgrowth_py import fpgrowth
>
> itemSetList = [['eggs', 'bacon', 'soup'],
>                ['eggs', 'bacon', 'apple'],
>                ['soup', 'bacon', 'banana']]
>
> freqItemSet, rules = fpgrowth(itemSetList, minSupRatio=0.5, minConf=0.5)
> print(rules)
> [[{'eggs'}, {'bacon'}, 1.0],
>  [{'bacon'}, {'eggs'}, 0.67],
>  [{'bacon'}, {'soup'}, 0.67],
>  [{'soup'}, {'bacon'}, 1.0]
> ]
> ```

Different frameworks provide different mechanisms to persist the generated rules, and you are free to use any format. In Python, the `pickle` module is a popular choice to store complex objects.

> In this project we do not specify *how* you should make the predictions so that students are free to devise their own recommendation system, but the association rules provide a starting point for many intuitive recommendations methods.

2. REST API Server

We will also implement a server that receives requests for playlist recommendations through a REST API. Students are free to choose the framework and language used to build the REST API. In Python, Flask or Fast API are widely used to expose a REST interface and are a good fit for a microservice architecture. The server should expose a REST endpoint at `http://<ip>:<port>/api/recommend` that responds to requests. A request will contain a list of songs the user likes, and the response should contain song recommendations.

You should use a specific port number for your REST API to avoid conflicts with the ports used by other students. A table with port allocations will be posted on the course website.

> The Flask Quickstart has instructions on how to launch a minimal Flask application. You can choose the `port` when running your Flask application by passing the `--port <number>` parameter to `flask run`. For example, user `ifs4` should use:
>
> ```
> export FLASK_APP=hello
> flask run --port 30502
> ```

> By default, Flask loads the application from `app.py`. If your file is in a different file, you should set the `FLASK_APP` environment variable (like in the previous example) to point to your file (do not include the `.py` extension as Flask expects a Python module name).
>
> In Flask, the path `api/recommend` at the end of the REST endpoint is called a *route*. Requests for a route are handled by a function, which should generate a response. You should annotate the function that will answer requests with `@app.route("/api/recommend")` in your Flask app (see also the section on routing in the Flask Quickstart).

Data in the request should be encoded in a JSON object containing a `songs` field with a list of songs. The response should also be a JSON object containing three fields: `songs` should be a list of songs that the user may enjoy, `version` should be a string indicating the version of your code currently running; and a `model_date` string indicating when the ML prediction model was last updated.

> Because requests will always carry some data, they should use the HTTP POST method. You can automate error handling in Flask by telling Flask to only accept POST requests by annotating your function with `@app.route("/api/recommend", methods=["POST"])`.
>
> Your function will need to parse the received JSON to extract the list of songs. You can use the `request` object to get the contents of a POST as JSON in two ways: you can access `request.json` if the request has a `Content-Type: application/json` header; or you can use `request.get_json(force=True)` to ignore the `Content-Type`.
>
> You should also return a JSON object. Because JSON-based interfaces are very common, your framework of choice might have facilities dedicated to this. In Flask, for example, the `jsonify` function can be called to build APIs with JSON.
>
> Your Flask application will need to load the ML recommendation model you created in the previous task. A detailed way of initializing Flask applications is described in the tutorial. In this assignment, your Flask application will need to load the recommendation model on startup and reload it whenever it changes. For example, the following two lines would initialize a variable `app.model` with the contents of a model saved using Python's `pickle` format:
>
> ```
> app = Flask(__name__)
> app.model = pickle.load(open("/path/to/model.pickle", "rb"))
> ```
>
> Note that the path to the file (`"/path/to/model.pickle"` in the example above) will need to be adjusted depending on whether your Flask front-end is running directly on the VM, inside the container, or using the Kubernetes shared volume (see below).

3. REST API Client

You will need a client that can send requests to the server for testing purposes. To evaluate your server, you can use songs from the `2023_spotify_songs.csv` dataset to generate requests.

You should develop a client that can generate playlist recommendation requests for any number of songs passed by parameter. For instance, users may want a playlist recommendation based on the information of just one song if they want to discover new artists or based on several songs they like to get similar songs aligned with their taste.

Students are free to implement the client in any way. A CLI client issuing an HTTP to the REST API is sufficient; for example, you can use tools like `wget` and `curl`. Alternatively, a Web-based front-end will also be accepted and will be graded with 1 point of extra credit.

> You can send a well-formed POST request to the server using the following `wget` call. The output will be written to a file called `response.out`. Note that this command sends the request to the server running on port 30502, you should change the port number to send the request to your application's port:
>
> ```
> wget --server-response \
>    --output-document response.out \
>    --header='Content-Type: application/json' \
>    --post-data '{"songs": ["Yesterday", "Bohemian Rhapsody"]}' \
>    http://localhost:30502/api/recommender
> ```
>
> Your client can be developed as a wrapper around this `wget` call.

Part 2: Continuous Integration and Continuous Delivery

In this project we will also employ CI and CD technologies. This part of the project does not require much coding, if any. Instead, it requires you to correctly configure a CI/CD pipeline to automatically build your containers and deploy them on the cluster. The requirements related to CI/CD in this assignment are as follows, but combining additional tooling or alternate solutions that exercise CI/CD are welcome and may be worth additional credit. Discuss any ideas with the instructor.

1. Create Docker Containers

You should create two Docker containers. One is the *ML container*, which will generate rules for the recommendation model and save them (e.g., using the `pickle` module). The other is the *frontend container* which will read the ML model and make it accessible through a REST API (e.g., by running Flask). Users will send requests to the frontend container when using the playlist recommendation service. This involves writing two Dockerfiles, one for each container.

The ML container should only contain the rule generation *code*, it should *not* include the dataset. Similarly, your frontend container should include only the REST API server *code*, it should *not*

include the generated rules for playlist recommendation. (We will discuss below how each container will access their data.)

> There are several tutorials on how to build a Flask application with Docker, with varying degrees of detail and advanced features. However, Docker's own documentation on building Python containers uses Flask as an example and is enough to get us started.
>
> Your REST API server might contain three files: a Dockerfile, a `requirements.txt` file (which contains the Python dependencies for your project), and the Python code to run the Flask app. Your Dockerfile could be based off of one of Python's base images, like `python:3.9-slim-bullseye`, which uses Debian Bullseye as the baseline system. On top of the base image, your Dockerfile should `pip3 install` packages from your `requirements.txt`, and copy your Python code inside the container. Finally, you should set Flask as the default application when your container starts by setting either `ENTRYPOINT` or `CMD` in your Dockerfile. The tutorial discusses each of these steps in detail. If you need to set environment variables in your Dockerfile (e.g., `FLASK_APP`), use the `ENV` command.
>
> Note that, by default, Flask listens for connections on the *localhost* address (`127.0.0.1`), which, in the case of a Docker container, is accessible only from *within* the container. To make your server reachable from outside the container, we need two things:
>
> 1. Make Flask listen on all addresses inside the container, including the address used to communicate with the outside world, by passing `--host=0.0.0.0` as a parameter to Flask in your `ENTRYPOINT` or `CMD`. (The `0.0.0.0` address is a special value which means "any address on this container".)
>
> 2. When running your container, forward traffic for your server's `port` into your container, so your application inside the container actually receives it. Use Docker's `--publish hostPort:containerPort` flag when running your container. `hostPort` should be the port traffic arrives at on the host; in user's `ifs4` case it is `30502` (see table above). `containerPort` should be whatever port Flask is listening on inside your container, which is `5000` by default, but you can change that by passing the `--port` parameter to Flask in your `ENTRYPOINT` or `CMD`.
>
> After you have built your image and run your container, you can test your server is responding by running `wget` as described above.

Your Docker image must be sent to a public hosting service where it can be downloaded by Kubernetes/ArgoCD; examples of such hosting services are DockerHub or Quay.io.

> When building your image locally, you can tag it and provide the repository where it will be stored using the `-t` option to `docker build`. A tag has the following format `[repository]/[name]:[version]`. In the example below, `repository` is `quay.io/cunha`, `name` is `playlists-recommender-system`, and the `version` is `0.1`. The `version` will be useful later as it is the means by which we will inform ArgoCD that our application has been

> updated. You can then manually push it to the repository:
>
> ```
> $ docker login quay.io
> ...
> $ docker build . -t quay.io/cunha/playlists-recommender-system:0.1
> $ docker push quay.io/cunha/playlists-recommender-system:0.1
> ```
>
> Docker image repositories may be configured to track a Git repository and automatically build new images as changes are pushed to the repository. If you set this up, pay attention to how the generated images are tagged, so you can retrieve individual versions later.

2. Configure the Kubernetes deployment and service

Write a YAML file specifying a Kubernetes deployment. A deployment specifies your application's pods (i.e., what containers each pod has) and how each should be deployed (e.g., the number of pod replicas). A deployment also specifies metadata that can be used to identify your application.

Write a YAML file specifying a Kubernetes service. A service specifies a publicly-accessible application. A service tells Kubernetes that requests to your `port` should be sent to your application's pods.

> In your deployment's template, you should add a label to allow identification of your pods. For example, you can add a label to the `.spec.template.metadata.labels` object, for example `app: <user>-playlist-recommender`. This will allow you to refer to your application's pods in the service definition. Your deployment's template should also specify that its containers use the Docker image you pushed to the public repository in the previous step.
>
> In your service's template's `.spec.selector`, you should select pods from your applications by using the label you generated in the deployment. You should also specify your container's port numbers inside `.spec.ports`.
>
> Once you have built both the deployment and service files, you can test everything is in order by running:
>
> ```
> kubectl -n <username> apply -f deployment.yaml
> kubectl -n <username> apply -f service.yaml
> kubectl -n <username> get deployments
> kubectl -n <username> get services
> ```
>
> You can use the files under this repo for example deployment and service files. The ArgoCD examples directory has many other examples.

You can then send recommendation requests to your application by using `wget` or `curl` and using the IP address listed as the service's `cluster-ip` as the destination IP, for example (assuming the `cluster-ip` is `67.159.94.11` and `port` is `30502`):

```
wget --server-response \
    --output-document response.out \
    --header='Content-Type: application/json' \
    --post-data '{"songs": ["Yesterday", "Bohemian Rhapsody"]}' \
    http://67.159.94.11:30502/api/recommender
```

Although students will deploy services in Kubernetes indirectly, using YAML files and ArgoCD, some Kubernetes commands can be useful to check and debug applications. The table below summarizes some useful Kubernetes commands. The `<namespace>` is given by the student's username on the cluster. The `kubectl` Cheat Sheet lists many other useful commands.

Command	Description
`kubectl -n <namespace> get pods`	List all pods in namespace
`kubectl -n <namespace> get service`	List all services in namespace
`kubectl -n <namespace> get deployment`	List all deployments in namespace
`kubectl -n <namespace> get pvc`	List all persistent volume claims in namespace
`kubectl -n <namespace> exec -it <pod-name> -- bash`	Get a shell inside the container
`kubectl -n <namespace> logs <pod-name>`	Get a pod's logs

After you are done testing your code in Kubernetes, be sure to delete the deployment and service using the following commands before configuring your application in ArgoCD in the next step:

```
kubectl delete deploy <deployment-name>
kubectl delete service <service-name>
```

Sharing the Model over a Persistent Volume

Applications deployed on Kubernetes are often stateless, meaning that the applications do not rely on any previously saved data in the cluster to function properly. However, there are scenarios where you need to deploy applications that collect and store data that must be kept after service shutdown or data that is shared across applications. Kubernetes PersistentVolumes

provide persistent storage for your containerized applications.

A persistent volume (PV) is a piece of storage provided by an administrator in a Kubernetes cluster. When a developer needs persistent storage for an application in the cluster, they request that storage by creating a persistent volume claim (PVC) and then mounting the volume to a path inside the pod. Once that is done, the pod claims any volume that matches its requirements (such as size, access mode, and so on). An administrator can create multiple PVs with different capacities and configurations. It is up to the developer to create a PVC for storage, and then Kubernetes matches a suitable PV with the PVC.

In this project, we provide a PersistentVolume to each student using the following configuration:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  namespace: <login>
  name: project2-pv2-<login>
  labels:
    type: local
spec:
  capacity:
    storage: 1Gi
  storageClassName: default-storage-class-<login>
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Recycle
  hostPath:
    path: /home/<login>/project2-pv2
```

> Note: You do **not** need to create a `PersistentVolume`, you need to create the `PersistentVolumeClaim`. To avoid multiple misconfigured `PersistentVolumeClaim`s from accumulating and using volumes reserved for other students, add a `selector` to your `PersistentVolumeClaim` YAML definition. Your definition should look like the following:
>
> ```
> apiVersion: v1
> kind: PersistentVolumeClaim
> metadata:
>   ...
> spec:
>   selector:
>     matchLabels:
>       namespace: <login>
>   ...
> ```

Each `PersistentVolume` is already configured inside the home of each student at `/home/<login>/project2-pv2` folder, using the Kubernetes's name of `project2-pv2-<login>` and one

distinct `storageClassName` per student given by `default-storage-class-<login>`. Each PV has a maximum of 1GB capacity and has the ability to allow many pods to write and read at same time. Your job here is to configure your deployments to be able to read and write files inside this folder.

The general idea is that the ML container should run whenever the dataset is updated (either manually, when triggered by the continuous delivery tool below), generate a new model and save it in the (shared) persistent volume claim. After that, the front-end container, responsible to generate the playlist recommendations, must watch the file containing the model inside the shared PVC. The front-end container can check for update by file modification date or changes in its checksum (e.g., MD5 or SHA). After a change is detected, the container must reload the model, performing recommendations using the most up-to-date version of the model. You should consider using a Kubernetes Job to run the ML container, as it does not need to run continuously. (Note, however, that if you create a Job, it needs to change name on each different execution of ArgoCD will get confused about the state of the job.)

3. Configure Automatic Deployment in ArgoCD

Automate the deployment of your application in ArgoCD. Push the files containing the specifications of your deployment and service (previous step) to a Git repository. Then configure ArgoCD to monitor this repository and automatically deploy and update your application on the cluster whenever the files change. Note that your repository should point ArgoCD to changes in (i) the Kubernetes configuration (deployment and service files), (ii) code changes (identified by the tag in the Docker image), and (iii) dataset updates (identified by some property of the dataset like the date).

> Before following these steps, be sure to log in to the ArgoCD server and change your password, as described in the "Cluster Specifics" section below.
>
> You can create your application over the GUI or over the command line. Here are some comments on some important fields; we give their name in the Web interface and the respective command-line parameter:
>
> - Repository URL (`--repo`): This should point to a public Git repository where you have descriptions for your Kubernetes deployment and service.
> - Path (`--path`): This should be the directory path of where the deployment and service files are located (if the deployment and service files are in the root directory, use `.` as your path).
> - Cluster URL (`--dest-server`): This should be `https://kubernetes.default.svc` , which is the path to our Kubernetes cluster.
> - Namespace (`--dest-namespace`): This refers to the Kubernetes namespace, and should be equal to your username on the cluster.

> - Select automatic sync to automatically deploy your application on the cluster (`--sync-policy auto`).
>
> Here is an example complete invocation on the CLI (similar values would be used on the Web interface):
>
> ```
> argocd app create guestbook \
>       --repo https://github.com/argoproj/argocd-example-apps.git \
>       --path guestbook \
>       --project $USER-project \
>       --dest-namespace $USER \
>       --dest-server https://kubernetes.default.svc \
>       --sync-policy auto
> # The shell will replace $USER with your username.
> ```

In order to execute your project using ArgoCD, the Git repository must contain, at least, the Kubernetes deployment code and a pointer to the dataset. You may create additional JSON or YAML files to store pointers to the dataset (e.g., a URL) and a Docker image tag. Note that the pointer to the dataset must not be hardcoded in the ML container's image; it should be passed by an environment variable (see Kubernetes and Docker documentations) or some other general mechanism.

Automatically Updating Rules when the Dataset Changes

One of the tests in Part 3 involves changing the configuration of your deployment to use a different playlist datasets (e.g., changing from `ds1` to `ds2`). This is made somewhat complicated as Kubernetes does not allow updating a Pod's variables after it has been created. As a result, ArgoCD fails to apply Deployment configuration changes and update the ML container's Pod unless we `force` the `replace` ment of the original Pod, which is not well support as of 2023 (see these issues for details).

One well-supported approach to sidestep this issue is to change your Pod's name (given by the `metadata.name` field in the YAML file) every time the playlist is updated. This will create a new Pod instead of reusing the existing Pod. This is sufficient to have continuous delivery of model updates by changing the deployment's YAML file. One complement is to configure ArgoCD to automatically prune the ML containers when they exit successfully by configuring automatic pruning of resources.

> Remember that creating a Pod is a lightweight operation and that the overhead will be dominated by the container execution, which is necessary in any case. So the overhead incurred by changing the Pod's name is not a lot more expensive.
>
> Other approaches to configure automated updates to the model when the deployment file changes are possible and will be accepted as valid solutions. Feel free to explore and evaluate alternate approaches. Particularly interesting or well-integrated solutions may be

> awarded extra-credit during grading.

Part 3: Exercise and Evaluate Continuous Delivery

Perform tests on the CI/CD infrastructure and write a discussion in a PDF to submit on Sakai. In particular, test that ArgoCD redeploys your container when you update (i) your Kubernetes deployment (e.g., increase or reduce the number of replicas), (ii) your code, and (iii) the training dataset. Measure how long the CI/CD pipeline takes to update the deployment by continuously issuing requests using your client and checking for the update in the server's responses (either the version or dataset date). Estimate if and for how long your application stays offline (inaccessible) during the update.

> There are many ways to trigger the automated deployment when using continuous delivery, and they depend on how the code is implemented as well as how the repositories are set up. Here is a straightforward way that should apply to most implementations of the recommendation system:
>
> 1. Update the number of replicas in your Kubernetes deployment. This should trigger an update in ArgoCD.
> 2. Update the dataset of playlists used to generate the recommendation model. This should trigger ArgoCD to re-run your ML container, which will then generate a new set of rules and send them to the frontend container.
> 3. Update one of the container images (for example, by updating the code version).
>
>   If building images manually, increment the version number when pushing it to DockerHub or Quay.io (note the `:0.2` instead of `:0.1` we used above):
>
>    ```
>    docker build . -t quay.io/cunha/playlists-recommender-system:0.2
>    docker push quay.io/cunha/playlists-recommender-system:0.2
>    ```

End-to-end Overview

The complete workflow covered in the project is shown in Figure 2. Red arrows represent steps that are done by the student; the green arrow represents the service's use after deployment (when it goes online); and the yellow arrows are steps performed automatically or indirectly by the CI/CD infrastructure when the code changes or an updated prediction model is pushed to the Git repository.

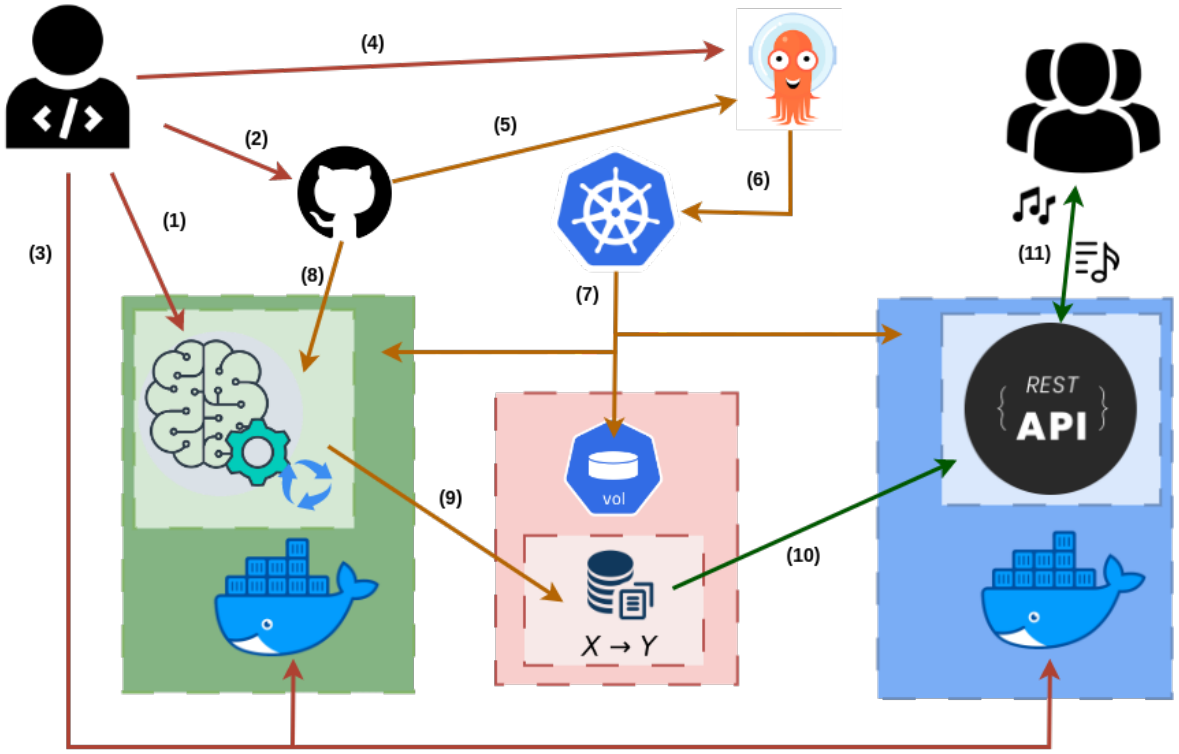

Figure 2. Development workflow.

1. Students will generate a microservice to build a recommendation model using a machine learning framework.

2. Students will create a public repository to maintain the service metadata (the dataset, code version, playlist, Kubernetes YAML file). For each new update in this repository (for example, an update to the dataset), CI/CD will be triggered and will update the Kubernetes deployment using the up-to-date metadata.

3. Students will write two Dockerfiles to build a container to (i) run the application responsible for generating a new recommendation model and (ii) run a REST API front-end for the recommendation service. Both containers will be uploaded to a hosting platform. Both containers will share a volume where the generated model will be saved (by the first container) and used by the recommendation service (the second container).

4. After creating the model, the Git repository, and both Docker images, and the Kubernetes deployment, students will configure the ArgoCD deployment. Students must deploy their services in their own Kubernetes namespace (defined by the login name) and in their own ArgoCD project area (defined by login name plus the suffix `-project`).

   > Steps (5) to (7) are performed indirectly when ArgoCD finds a new update in the repository configured in step (4). While steps (8) and (9) are performed by the ML container when it detects a new version of dataset.

5. ArgoCD will be in charge of detecting a new update in the configured repository, e.g., a new Docker image version. When an update is detected, ArgoCD will pull the Kubernetes specs and subsequent changes from the Git repository and update the deployment to the up-to-

date configuration.

6. ArgoCD sends a request to Kubernetes to create or update a service.
7. Kubernetes downloads the image from the hosting platform if it is not present in the cluster. If the image is already present, it will only check if it has any update. Also, Kubernetes creates the services in pods using the resources defined in the YAML file and the persistent volume used to share files between pods.
8. In a real scenario, the dataset can change over time (i.e., new playlists can be added, modified or removed), so both applications must be prepared to handle these events. In the case of the ML container, ArgoCD should detect that a new dataset version is available in the repository and trigger execution of the ML container to update the model.
9. This new set of rules will be saved on a shared folder also accessible by the front-end container.

> Steps (10) and (11) are performed when a user sends a recommendation request to the REST API.

10. Once the front-end container is started, students can send requests to their service's address.
11. The recommendation service must always use the newest model version in the recommendation process.

Cluster Specifics

This programming assignment does not require a significant amount of CPU, RAM or DISK, so you are free to pursue its development on your own machine. However, note that configuring Kubernetes and ArgoCD is complicated; unless you are willing to invest some effort in setting these frameworks up, consider doing this part of the assignment on the cluster VM.

ArgoCD CLI/UI

Each student has an account to interact with ArgoCD via CLI or its Web service running at port `31443`. The default password is the concatenation of your `username` and the MD5 hash of your private key (without including the trailing `\n` newline character). Remember to update this password on your first login. To log in and update the password via the CLI, execute the commands below. It is also possible to change the password via the Web interface in the "User Info" section.

```
argocd login localhost:31443 --username <username> --password <password> --insecure
argocd account update-password
```

To compute your password on the CLI, you use one of the two approaches below:

```
cat .ssh/id_rsa.pub | tr -d "\n" | md5sum
echo -n "<contents of public key>" | md5sum
```

Students can create applications using the CLI or Web interfaces. However, the command line interface can also be used to submit or to debug applications. The table below summarizes some useful commands. The ArgoCD documentation has many use case examples and provides a tutorial to getting started.

Command	Description
`argocd login SERVER [flags]`	Log in to Argo CD
`argocd app create APPNAME [flags]`	Create an application
`argocd app delete APPNAME [flags]`	Delete an application
`argocd app sync APPNAME [flags]`	Sync an application to its target state
`argocd proj list`	List projects
`argocd app logs APPNAME [flags]`	Get logs of application pods

Accessing the REST API server

Kubernetes will deploy your REST API server on the cluster. Once Kubernetes allocates a `host` and `port` to your server, you can access it in two ways.

First and easiest, you can access it by generating REST requests from the command line from within the cluster. This can be done, e.g., using `curl` or `wget`.

Second, you can access it by generating REST requests from a remote machine, but piping these machines through an SSH tunnel, so they can reach the REST API server through the tunnel. This is necessary due to the network security policy on our cluster. The following command will forward all data arriving at a given `<local-port>` on your machine to a `<k8s-port>` on the cluster's main VM. Both `<k8s-host>` and `<k8s-port>` must be the same one allocated to your server by Kubernetes.

```
ssh -fNT -L <local-port>:<k8s-host>:<k8s-port> \
         <username>@vcm
```

After the SSH tunnel is established, you can make REST requests to your server from your machine using `<local-port>`.

Troubleshooting

When debugging your REST API server, test it first by directly running it on the host machine with `flask run` or calling your module as a script. After you have tested your server on the host, test it inside Docker before you test deployment on Kubernetes. When testing under Docker there are common things to check:

- Check the port numbers . The Flask app runs inside the container on port X (given in `run(..., port=X)` or `flask run --port X`). Then, we forward port Y on the host to port X inside the container by executing `docker run --publish Y:X` . Finally, we can access the server by running `wget http://localhost:Y` . In this setup, `wget` will send a request to port Y on the host; Docker will have configured the network to forward our request to the container on port X; our server will receive the request and send a response. (Flask's default port number is `X = 5000` .)
- Check that Flask is listening on all ports inside the container. You should set `run(host="0.0.0.0")` or `flask run --host 0.0.0.0` when launching your Flask app inside the container. After testing on Docker, try deploying on Kubernetes. Here are some suggestions:

After checking under Docker, try your deployment in Kubernetes. Again, check that the port numbers in your deployment and service files match. Also check that your deployment's pods are in the `running` state. Pods may crash and enter a back-off state. Common errors include `ImagePullBackoff` , which happens when Kubernetes cannot download your container's image. This may happen, for example, because your container image is private or because the path to the container image is incorrect in the deployment file. Another error is `CrashLoopBackoff` , which happens when your container crashes on startup. This should not happen after successfully testing under just Docker, but if it does happen, you may need to rebuild and update your container image on the repository (DockerHub or Quay.io).

What to Submit

You should submit on Sakai:

1. The ML code responsible for generating association rules and a version of your model.
2. The Flask code responsible for running the Web front-end.
3. The client application, scripts, or Web front-end in charge of demonstrating access to your REST API.
4. 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.

5. The YAML files describing the Kubernetes deployment and service.

6. 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`.

7. PDF file with introduction and discussion of the following points:

    - Discussion of test cases performed with Kubernetes and ArcoCD to exercise continuous integration functionalities and expected results.
    - 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.
    - How you detect model changes in the REST API back-end.
    - How your containers obtain the new dataset when regenerating the model.