Project 2: DevOps and Cloud Computing

Introduction

DevOps is the combination of practices and tools that increases an organization's ability to deliver applications and services at high velocity: evolving and improving products at a faster pace than 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 delivery. Figure 1 exemplifies the activities in Continuous Integration (CI) and Continuous Delivery (CD), the two core concepts of the DevOps methodology.

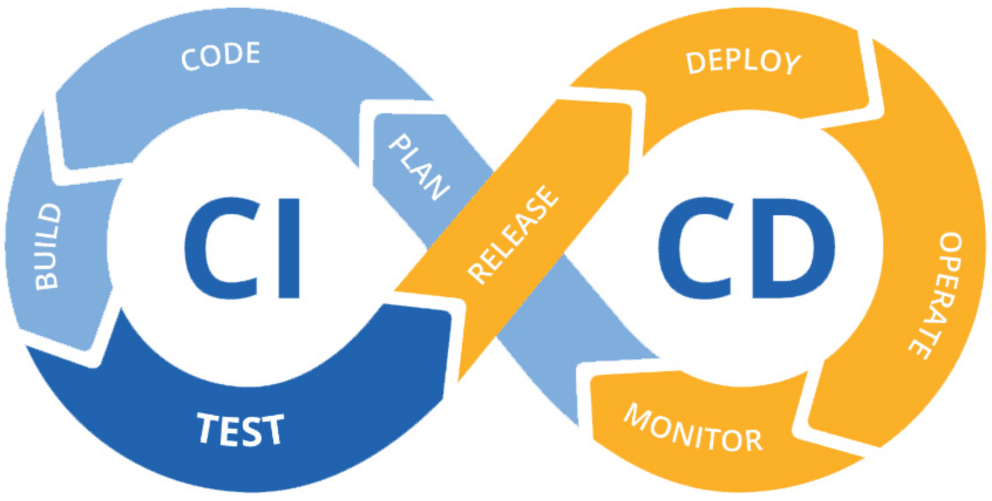

Figure 1. DevOps Pipeline. Source: Cansin Acarer.

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* is the continuation 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 machine learning application built on microservices using continuous delivery practices. This practice has also been referred to as MLOps using a microservice. The student will obtain experience by using some of the most popular tools in this context: Docker to conteinerize 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 tool for Kubernetes.

Students will have the task to create a machine learning (ML) application to predict whether a Twitter user is American or not based on the text of a single tweet. (You are free to train and test with predictors for other languages as well, although classification may be less accurate because of fewer data available for other languages and less challenging due to fewer countries sharing a common language. You are also free do train different predictors altogether.)

Dataset

We will use a sample of the Covid-19 dataset used in Project 1. The dataset sample is available on HDFS under `hdfs://vcm-23691:9000/datasets/covid19-sample`. It includes two parts:

1. The `training.csv` file contains 10 thousand random records to be used in the ML training phase.
2. The `test.csv` file contains 30 thousand records to be used in the prediction step.

Both samples contain tweets in English (`lang` equal to `en`) where the `country_code` is not null. To retrieve the dataset files from HDFS, you can use the HDFS CLI command like: `hdfs dfs -get /datasets/covid19-sample/training.csv <outputFolder>`.

Software Components

We will implement an application to expose our predictor using microservices. The application will have three components described next. These software components are modular and do not require significant resources to develop. Students are free to prototype on their machines or on `vcm-23691`.

ML model to predict if a Twitter user is American

We will develop code to train of an ML model for the predictor. More specifically, we will train a binary classifier model capable of predicting if the author of the tweet is American based on the tweet (`text` column). We define that a tweet was authored by an American if the `country_code` column is equal to `US`. The predictor should output `1` if the author is inferred to be American and `0` otherwise.

A common way to implement such a classifier is performing a cleaning step in the tweet text, converting the text into a vector of features, then use the feature vector as input to the ML classifier.

1. *Cleaning up text* involves tokenizing (splitting words) and removing stop words (e.g., "the"). This can be performed, for example, using Scikit-Learn's `CountVectorizer` class by using the `stop_words="english"` parameter.
2. *Converting the text into a vector of features* is often the most important step in training an ML model. In this course, any reasonable approach will be sufficient. Suggestions include the Bag of Words and TF-

IDF approaches, both available in Scikit-Learn. The Bag of Words approach is provided directly by the `CountVectorizer` class, and TF-IDF can be computed by the `TfidfTransformer` class.

3. *Training the ML classifier* will use the `training.csv` dataset, where the ML framework will get both the tweet's `text` feature vector (input) and the `country_code` (output), allowing it to capture the relationship between them.

Students are free to use whatever ML framework they want. The Python Scikit-Learn and Pandas frameworks are already installed in our cluster and are sufficient for completing the assignment. This tutorial discusses how to build a predictor based on text using Scikit-Learn; you should be able to train a classifier by following the steps under for extracting features from text files and training a classifier. Optionally, you can also evaluate the accuracy of your predictor following the steps for evaluating the performance using the test set; a performance of around 75% is achievable with the Naive-Bayes classifier in the tutorial and is enough for the purposes of this assignment.

> The tutorial builds a predictor to identify which newsgroup a post is form. Newsgroups are similar to discussion forums where users can post messages.
>
> The tutorial uses a dataset different from ours, but the data read from the "20newsgroups" dataset used in the tutorial is captured in two lists:
>
> - `twenty_train.data` is a list of strings, where each element contains the content of a newsgroup post.
> - `twenty_train.targets` is a list of strings, where each element is the newsgroup the corresponding post is from.
>
> For our dataset, we should build a list strings (`tweets`) containing tweet texts and another list (`targets`) of integers containing `0` or `1` indicating whether each tweet is by an American. One important invariant is that we need `tweets` and `targets` to be ordered; that is, `tweets[i]` and `targets[i]` must refer to the same tweet for all values of `i`.

Different frameworks provide different mechanisms to persist the ML prediction model, and you are free to use any format. In Python, the pickle format is a popular choice to store complex objects. After following the steps in the tutorial, you can save in a pickle either the `Pipeline` or the trained `MultinomialNB` class.

REST API server*

We will also implement a server that makes the predictor accessible 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. The server should expose a REST endpoint at `http://127.0.0.1:port/api/american` that responds to requests containing the text of a tweet. You should use a port number given by the table below.

ID	Port
xc139	5001
rf96	5002
zf44	5003

ID	Port
jh678	5004
rj133	5005
kl328	5006
lmp81	5007
ifs4	5008
jt304	5009
zt40	5010
yw345	5011
yy240	5012
cz155	5013
hz185	5014
xz249	5015
yz605	5016
zz188	5017
tz88	5018

> 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 5008
> ```
>
> By defualt, Flask loads the application from `app.py`. If your file is in a different file, you should set the `FLASK_APP` environment variable like above to point to your file (do not include the `.py` extension as Flask expects a Python module name).
>
> In Flask, the path `api/american` 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 use annotate your function that will answer resquests with `@app.route("/api/american")` 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 `text` field with the tweet's text. The response should also be a JSON object containing three fields: `is_american` should be `1` if the model predicts the author of the tweet to be American and `0` otherwise; `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 should 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/american", methods=["POST"])`.
>
> Your function will need to parse the received JSON to extract the tweet's text. 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 prediction model you created in the previous step. A detailed way of initializing Flask applications is described in the tutorial, but you can directly add the model directly as a field of the Flask application:
>
> ```
> app = Flask(__name__)
> app.model = pickle.load(open("/path/to/model.pickle", "rb"))
> ```

REST API client

You should also prepare a client that can send requests to the server for testing purposes. To evaluate your server, you can use tweets from the `test.csv` dataset in HDFS to generate requests and compute the accuracy of predictions.

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 5% extra credit.

> You can send a well-formed POST request to the server using the following command. The output will be written to a file called `response.out`. Note that this command sends the request to the server running on port 5008 (user `ifs4`), 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 '{"text": "#covid19 new york"}' \
>     http://localhost:5008/api/american
> ```

Continuous Integration and Continuous Delivery

In this project we will also employ CI and CD technologies. Our suggestion is that you add CI and CD to your project early on, so you can take advantages of the automation while building your solution.

The requirements related to CI/CD in this assignment are as follows. Combining additional tooling or alternate solutions that exercise CI/CD are welcome, and may be worth additional credit. Discuss any ideas with the instructor.

Create Docker container

Create a Docker container with the REST prediction service. This involves writing a Dockerfile to create the container with the REST API server.

Your Docker image should contain only the REST API server *code*. It should *not* include the ML prediction model. You should either download the model at runtime when the REST API server starts (e.g., from a URL) or read the model from a volume attached to your container.

> 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 documention 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 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 machine".)
> 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 `5008` (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.

> Docker image repositories may be configured to track a Git repository and automatically build new images as changes are pushed to the repository.

> When building your image, you can tag it and give 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 `american-predictor`, 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/american-predictor:0.1
> $ docker push quay.io/cunha/american-predictor:0.1
> ```

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 specified 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][kubernetes-service-def].

> 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: compsci401-sp22-predictor`. 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, you should select pods from your applications by using label you generated in the deployment in `.spec.selector`. 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 apply -f deployment.yaml
> kubectl apply -f service.yaml
> kubectl get deployments
> kubectl 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 prediction 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 `10.106.100.175` and `port` is `5008`):

> ```
> wget --server-response \
>     --output-document response.out \
>     --header='Content-Type: application/json' \
>     --post-data '{"text": "#covid19 new york"}' \
>     http://10.106.100.175:5008/api/american
> ```
>
> 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 commands in this application. 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 your namespace |
> | `kubectl -n <namespace> get service` | List all services in your namespace |
> | `kubectl -n <namespace> get deployment` | List all deployments in your 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:
>
> ```
> kubectl delete deploy <deployment-name>
> kubectl delete service <service-name>
> ```

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.

> 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 in parenthesis the 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 american-predictor \
>       --repo https://github.com/cunha/cloudcomp-argocd-k8s-predictor \
>       --path . \
>       --project cunha-project \
>       --dest-namespace cunha \
>       --dest-server https://kubernetes.default.svc \
>       --sync-policy auto
> ```

Make tests to exercise and evaluate continuous delivery

Perform some 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 your container. Estimate how long the CI/CD pipeline takes to run by issuing requests using your client and checking for the update in the server's responses. Estimate 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 and the repositories set up. Here is a straightforward way that should apply to most implementations of the predictor app:
>
> 1. Update the version number in the responses generated by the predictor.
>
> 2. Update your container image and tag/version.
>
>   For example, 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/american-predictor:0.2
>    docker push quay.io/cunha/american-predictor:0.2
>    ```
>
> 3. Update the deployment YAML file to use the new container image by changing the `.spec.template.spec.caontiners[0].image` key to point to the new container image. For example: `image: quay.io/cunha/american-predictor:v0.2`
>
> You can mostly skip step two above if you configure DockerHub or Quay.io to automatically build images for your containers. By default, DockerHub and Quay.io automatically tag new images with the last commit's short hash, so you can use the short hash instead of `0.2` in your deployment file!

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 (i.e., 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.

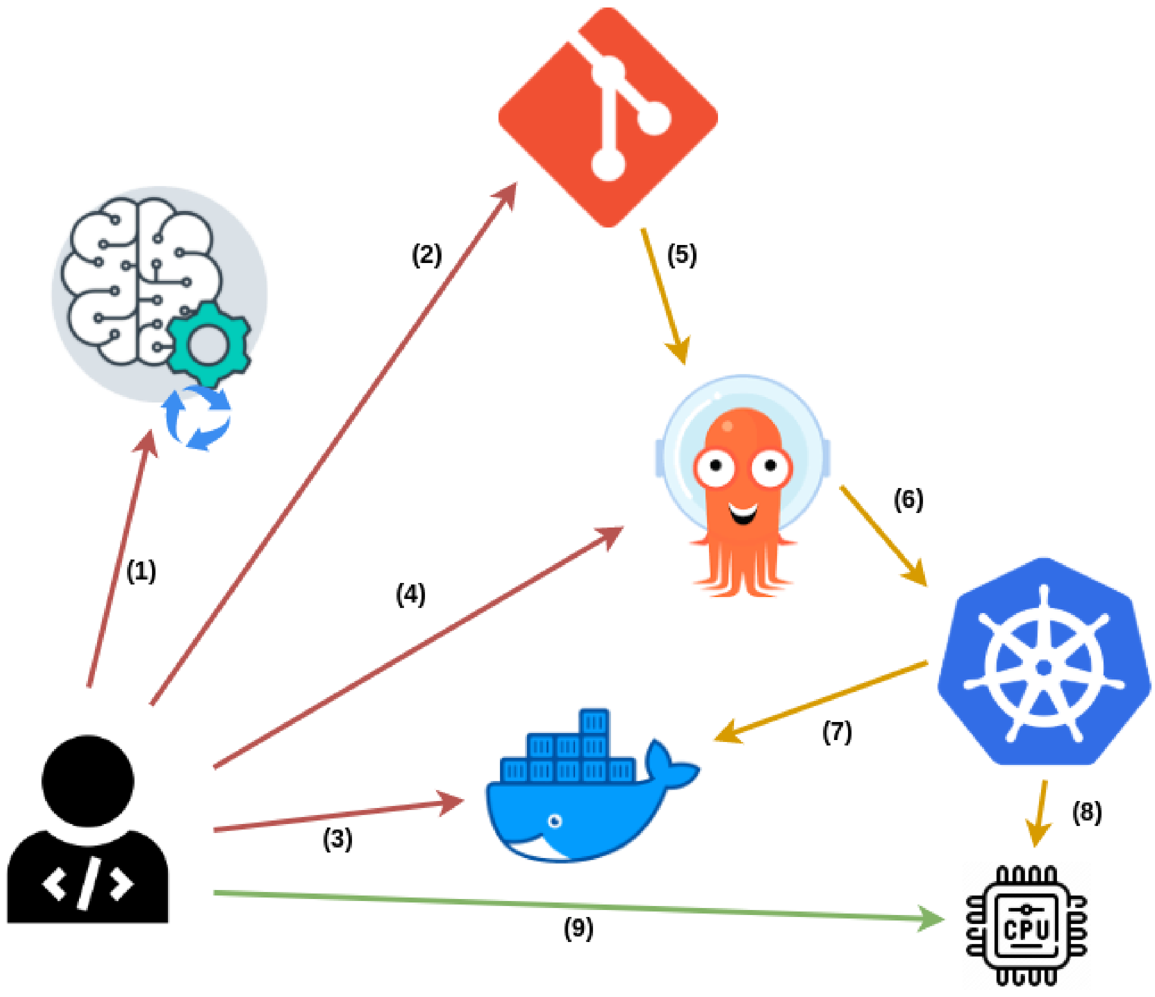

Figure 2. Development workflow.

1. Students will train an ML prediction model using the `training.csv` dataset.
2. Students will create a public repository to maintain their model and others application specs (Kubernetes YAML file). For each new update in this repository, e.g., a new ML model version, ArgoCD will be capable of updating the deployment using the new configuration.
3. Students will write a Dockerfile to build a container to run the prediction service. The container will be uploaded to a hosting platform.
4. After creating the model, the Git repository, and the Docker image, 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 (8) are performed indirectly when ArgoCD finds a new update in the repository configured in step (4).

5. ArgoCD pulls the Kubernetes specs and subsequent changes to the Git repository.
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.
8. Kubernetes creates the service in a pod using the resources defined in the YAML file.
9. Once the service is started, students can send requests to their service's address.

Cluster Specifics

ArgoCD CLI/UI

Each student has an account to interact with ArgoCD via CLI or web-service (running at port `32535`). The default password is the concatenation of your `username` and `123456`, 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:32535 --username <username> --password <password> --
insecure
argocd account update-password
```

Students can create applications by using only the Web interface. However, the command line interface can also be used to submit or to debug applications. The table below summarizes some useful commands. The ArgoCD's 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, 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-23691.vm.duke.edu
```

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

What to Submit

You should submit on Sakai:

1. The ML training code and a version of your model.
2. The Dockerfile to build your container in charge of running the REST API, and any additional code required to build your container image.
3. The YAML file describing the Kubernetes deployment.
4. The YAML file describing the ArgoCD application, exported from ArgoCD's UI/CLI interface.
5. The client application, scripts, or Web front-end in charge of demonstrating access to your REST API.
6. A PDF file discussing the tests performed in Kubernetes and ArgoCD to exercise the continuous integration functionalities. Discuss at least how long it takes for changes in your code to be deployed and whether the application stays offline.