# Cloud/DevOps Curriculum Series for Online Learning

# Lab 1 – Kubernetes on PRP Nautilus with Visual Cloud Computing app

**Technical Contacts:** Dr. Prasad Calyam (calyamp@missouri.edu); Dr. Songjie Wang (wangso@missouri.edu)

*Release 1: May, 2021*

## Contents

## 1 Purpose of the Lab

In this lab, you will learn Kubernetes knowledge and skills by using the PRP Nautilus cloud platform. You will perform hands-on activities on the fundamentals of Kubernetes, e.g., Pod, Deployment, Service, Load Balancing, Ingress, and Persistent Volumes, deploy an example stateless application to learn how to deploy application without persistent data storage, and deploy the a visual cloud computing (VCC) application to learn how to stream and analyze video data with object tracking algorithms on the cluster.

## 2 Prerequisites

- For you to be able to understand the concepts in this lab, we recommend that you have existing knowledge and experience with Docker containers. You could still follow along the lab without knowledge on Docker, but it would be difficult to understand how applications are running and communicating in Kubernetes cluster.
- You need to have a personal computer with command line access. We recommend that you use Linux/MacOS. If you are using Windows machine, we will provide link to help you install command line tools on it.

## 3 References to guide lab work

Please use the links below to learn more about the PRP Nautilus platform, Docker, and Kubernetes to help you understand the lab contents.

- Pacific Research Platform – *http://pacificresearchplatform.org/*
- Nautilus Kubernetes Cluster – *http://pacificresearchplatform.org/nautilus/*
- Docker – *https://docs.docker.com*
- Docker Hub – *https://hub.docker.com/*
- Kubernetes – *https://kubernetes.io/*
- Kubectl overview – *https://kubernetes.io/docs/reference/kubectl*
- Some of the essential concepts:
  - Pods: *https://kubernetes.io/docs/concepts/workloads/pods/*
  - Deployment: *https://kubernetes.io/docs/concepts/workloads/controllers/deployment/*
  - Service: *https://kubernetes.io/docs/concepts/services-networking/service/*
  - Ingress: *https://kubernetes.io/docs/concepts/services-networking/ingress/*
  - Cluster DNS: *https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/*
  - PersistentVolumes: *https://kubernetes.io/docs/concepts/storage/persistent-volumes/*
  - StatefulSets: *https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/*

## 4 Overview

Kubernetes is a popular open-source system for automating the deployment, scaling, and management of containerized applications. In this lab, we will do some hands-on learning of Kubernetes fundamental knowledge and skills. We use the Nautilus, a Kubernetes cluster provided by the Pacific Research Platform (PRP) and supported by National Science Foundation. Nautilus is a HyperCluster for running containerized Big Data Applications. It is using Kubernetes for managing and scaling containerized applications and Rook for automating Ceph data services. Nautilus runs up-to-date versions of the open-source Kubernetes software, so you can use all the existing plugins and tooling from the Kubernetes community. Applications running on Nautilus are fully compatible with applications running on any standard Kubernetes environment, whether running in on-premises data centers or public clouds such as Amazon AWS or Google GCP. This means that you can easily migrate any standard Kubernetes application to other platforms without any code modification required.

## 5 Access Nautilus Account Web Portal

**Note**

This section is just for your information. We will not use this login method for this lab. Instead we will distribute a user namespace to you upon request. You will be able to interact with the cluster using the config file that you will receive for your individual namespace (we will send you the config file upon request).

Nautilus platform integrates CILogin to allow any users from participated institutions to login and use their platform. But to simplify the lab, we have created a temporary account for each student to practice the lab. To get access to the PRP Nautilus cluster, do the following:

- Point your browser to the PRP Nautilus portal *https://nautilus.optiputer.net/*
- On the portal page click on “Login” button at the top right corner
- On this page, select an Identity Provider from the menu by searching for ”Missouri” and select ”University of Missouri System”, and Click “Log On” button to use your existing credentials from Mizzou to sign in.
- You could also use your personal email (e.g. Gmail) to login and access the platform. Just select Google on the main login page and you will be able to log in.
- Once you login to the portal, you will need to submit individual request to join a project by sending email to the project admin.

## 6 Command Line Access to Nautilus Cluster

Kubernetes largely relies on command line operations to interact with the cluster resources as well as your application that will be deployed on the cluster. For this purpose, you will need to setup remote command line interface using the Kubernetes tools, Kubectl.

**Note**

It is highly recommended that you use Linux/MacOS for running the commands in this lab. On Windows you can use PowerShell, but some of the commands might be different from the lab instruction.

### 6.1 Install Kuberctl Cli tool

Kubectl allows you to run commands against Kubernetes clusters. You can use kubectl to deploy applications, inspect and manage cluster resources, and view logs. Follow the official Kubernetes documentation to install the Kubectl tool on your local laptop – *Kubectl Installation*. From this link, you can click the instruction links to install Kubectl on Linux, MacOS, or Windows.

### 6.2 Configure kubectl to access Nautilus

Once you have installed Kubectl tool, you will need to configure the Kubectl tool with your own information:

- We have worked with Nautilus technical support team to acquire a namespace for each of you to do this lab. The namespace and its related login information are enclosed in a configuration file named ”config”.

  **Note**

  Pay attention to the config file. Open the file with your preferred text editor and look for the line with “namespace: mizzou-nautilus-user-XXX”. The namespace will be YOUR namespace to interact with the Nautilus cluster.

- You will need to download your specific config file, re-name it into “config”, and place it in the “ /.kube” directory on your computer to be able to interact with the Nautilus Kubernetes cluster.

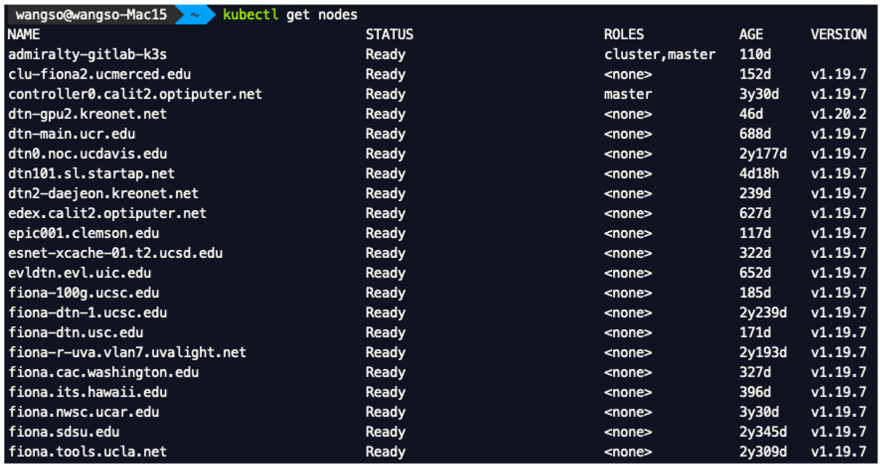
**Note**

This “.kube” folder should be there on your computer after you install kubectl tool. If it does not exist, then run the following command to create one.

```
$ mkdir ~/.kube
$ cp /your/path/to/downloaded/config ~/.kube/.
```

- Test to see if kubectl can connect to the cluster:

```
$ kubectl get nodes
```



```
 wangso@wangso-Mac15  ~  kubectl get nodes
NAME                              STATUS   ROLES            AGE     VERSION
admiralty-gitlab-k3s              Ready    cluster,master   110d
clu-fiona2.ucmerced.edu           Ready    <none>           152d    v1.19.7
controller0.calit2.optiputer.net  Ready    master           3y30d   v1.19.7
dtn-gpu2.kreonet.net              Ready    <none>           46d     v1.20.2
dtn-main.ucr.edu                  Ready    <none>           688d    v1.19.7
dtn0.noc.ucdavis.edu              Ready    <none>           2y177d  v1.19.7
dtn101.sl.startap.net             Ready    <none>           4d18h   v1.19.7
dtn2-daejeon.kreonet.net          Ready    <none>           239d    v1.19.7
edex.calit2.optiputer.net         Ready    <none>           627d    v1.19.7
epic001.clemson.edu               Ready    <none>           117d    v1.19.7
esnet-xcache-01.t2.ucsd.edu       Ready    <none>           322d    v1.19.7
evldtn.evl.uic.edu                Ready    <none>           652d    v1.19.7
fiona-100g.ucsc.edu               Ready    <none>           185d    v1.19.7
fiona-dtn-1.ucsc.edu              Ready    <none>           2y239d  v1.19.7
fiona-dtn.usc.edu                 Ready    <none>           171d    v1.19.7
fiona-r-uva.vlan7.uvalight.net    Ready    <none>           2y193d  v1.19.7
fiona.cac.washington.edu          Ready    <none>           327d    v1.19.7
fiona.its.hawaii.edu              Ready    <none>           396d    v1.19.7
fiona.nwsc.ucar.edu               Ready    <none>           3y30d   v1.19.7
fiona.sdsu.edu                    Ready    <none>           2y345d  v1.19.7
fiona.tools.ucla.net              Ready    <none>           2y309d  v1.19.7
```

You should see the above result that shows all the nodes included in the Nautilus Kubernetes cluster. If you do not see the output, re-configure your Kubectl with the right configuration file.

**To Submit**

You need to take a **screen shot** of the result for your lab submission to verify that you are able to connect to the Nautilus cluster

- Some useful commands to learn
  - List all the pods in your namespace (if you dont specify the -n option, kubectl will run the commands using your default namespace)

    ```
    $ kubectl get pods -n Your-NameSpace
    ```

    You can also try without -n option.
  - List all the deployments in your namespace

    ```
    $ kubectl get deployments -n Your-NameSpace
    ```

    You can also try without -n option.

– List all the services in your namespace

```
$ kubectl get services -n Your-NameSpace
```

You can also try without -n option.

## 7 Download the source git repo for the lab

We have prepared a git repository with all the source files for this lab. Please use your installed Git (either command line or desktop version) to download the repo onto your local machine:

```
$ git clone https://github.com/wangso/mizzou_nautilus_k8s_lab.git
```

**Note**

The remaining of the lab will be done inside the cloned git repo directory.

## 8 First deployment in Kubernetes

**Warning**

Containers are stateless. ALL your data WILL BE GONE FOREVER when container restarts, unless you store it in a persistent volume (we will touch this later in the lab).

### 8.1 Launch a simple pod

In this section, we will create a simple generic pod using a base Linux Docker image, and login into it.

#### 8.1.1 Create a pod definition file (.yaml file or manifest file)

**Note**

Indentation is important in manifest files, just like in Python. Please pay attention to all the indentation in your manifest files and do not break them, otherwise your resource will be created correctly. Also, do not use tabs, always use spaces to align the text. After you copy the code, make sure they all align the same way as in this manual.

From where you are in the git repository directory,

```
$ cd "your-path-of-repository"/basics
```

Open "test-pod.yaml" and examine the code in the file:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  containers:
  - name: mypod
    image: centos:centos7
    resources:
      limits:
        memory: 100Mi
        cpu: 100m
      requests:
        memory: 100Mi
        cpu: 100m
    command: ["sh", "-c", "sleep infinity"]
```

**Warning**

Nautilus platform requires that you always specify a minimum amount of resources (CPU, memory, etc) in your manifest files. Nautilus will continue to monitor your resources usage as opposed to the amount you requested, and will warn you and even temporarily lock your namespace if you are under-utilizing the resources. So please DO NOT try to increase the limits/requests in the manifest files to large amounts!

### 8.1.2 Create a pod and interact with it using kubectl commands

- From the “basics” directory, create a simple pod based on the manifest file:

  ```
  $ kubectl apply -f test-pod.yaml
  ```

- Check to see if the pod is there:

  ```
  $ kubectl get pods
  ```

- If it is not yet in running state, you can check what is going on with it using:

  ```
  $ kubectl get events --sort-by=.metadata.creationTimestamp
  ```

- Events and other useful information about the pod can be seen using describe command:

  ```
  $ kubectl describe pod test-pod
  ```

- If the pod is in Running state, let’s log into it

  ```
  $ kubectl exec -it test-pod -- /bin/bash
  ```

  You are now inside the (container in the) pod!

- We will want to check the status of the networking. But ifconfig is not available in the image we are using; so let’s install it

  ```
  $ yum install net-tools
  ```

- Now check the networking:

  ```
  $ ifconfig -a
  ```

- Exit the Pod (with either Control-D or exit). You should see the same IP displayed with kubectl

  ```
  $ kubectl get pod -o wide test-pod
  ```

- We can now destroy the pod

  ```
  $ kubectl delete -f test-pod.yaml
  ```

- Check that it is actually gone:

  ```
  $ kubectl get pods
  ```

- Now, let’s create it again:

```
$ kubectl apply -f test-pod.yaml
```

- Does it have the same IP?

  ```
  $ kubectl get pod -o wide test-pod
  ```

- Log back into the pod:

  ```
  $ kubectl exec -it test-pod -- /bin/bash
  ```

  Install again net-tools, and check the IP of the pod. What does the network look like now?

- Press Ctrl+D to exit from the pod

- Finally, let's delete explicitly the pod:

  ```
  $ kubectl delete pod test-pod
  ```

## 8.2 Launch a deployment

You saw that when a pod was terminated, it was gone. While above we did it by ourselves, the result would have been the same if a node died or was restarted. However, in a real-world application, in most cases we want to have a pod re-created if we accidentally delete it or if the pod dies for some reason. In order to gain a higher availability, the use of Deployments is recommended. With a Deployment, whenever a pod is deleted or died, the Deployment will automatically create a replacement pod that does the same job.

### 8.2.1 Create a deployment definition file (.yaml file)

From where you are in the git repository directory,

```
$ cd "your-path-of-repository"/basics
```

Open "test-deployment.yaml" and examine the code in the file:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-dep
  labels:
    k8s-app: test-dep
spec:
  replicas: 1
  selector:
    matchLabels:
      k8s-app: test-dep
  template:
    metadata:
      labels:
        k8s-app: test-dep
    spec:
      containers:
      - name: mypod
        image: centos:centos7
        resources:
           limits:
             memory: 500Mi
             cpu: 500m
           requests:
             memory: 100Mi
```

```
          cpu: 50m
      command: ["sh", "-c", "sleep infinity"]
```

### 8.2.2 Create a deployment and interact with it using the kubectl tool

- From the “basics” directory, create a deployment based on the manifest file:

  ```
  $ kubectl apply -f test-deployment.yaml
  ```

- See if you can find it:

  ```
  $ kubectl get deployments
  ```

- The Deployment is just a conceptual service, though. See if you can find the associated pod:

  ```
  $ kubectl get pods
  ```

- Once you have found its name, let’s log into it (replace the name of the pod with your own pod IDs))

  ```
  $ kubectl get pod -o wide pod_ID
  $ kubectl exec -it pod_ID -- /bin/bash
  ```

  You are now inside the (container in the) pod! Try various commands as before (e.g. install net-tools and check your network configuration).

- Press Ctrl+D to exit the pod when you are done

- Let’s now delete the pod!

  ```
  $ kubectl delete pod pod_ID
  ```

- Is it really gone?

  ```
  $ kubectl get pods
  ```

- What happened to the deployment?

  ```
  $ kubectl get deployments
  ```

- Get into the new pod

  ```
  $ kubectl get pod -o wide pod_ID
  $ kubectl exec -it pod_ID -- /bin/bash
  ```

  Was anything preserved?

- Let’s now delete the deployment:

  ```
  $ kubectl delete -f test-deployment.yaml
  ```

- Verify everything is gone:

  ```
  $ kubectl get deployments
  $ kubectl get pods
  ```

## 8.3 Load Balancing and Service

Orchestration is often used to spread the load over multiple nodes. In this section, we will launch multiple Web servers. To make distinguishing the two servers easier, we will force the node name into their homepages. Using stock images, we achieve this by using an initial container.

#### 8.3.1 Create a Deployment with multiple HTTP servers

From where you are in the git repository directory,

```
$ cd "your-path-of-repository"/basics
```

Open “test-http.yaml” and examine the code in the file:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-http
  labels:
    k8s-app: test-http
spec:
  replicas: 2
  selector:
    matchLabels:
      k8s-app: test-http                                              ①
  template:
    metadata:
      labels:
        k8s-app: test-http
    spec:
      initContainers:
      - name: myinit
        image: busybox
        command: ["sh", "-c", "echo '<html><body><h1>I am ' `hostname`
    '</h1></body></html>' > /usr/local/apache2/htdocs/index.html"]
        volumeMounts:
        - name: dataroot
          mountPath: /usr/local/apache2/htdocs
      containers:
      - name: mypod
        image: httpd:alpine
        resources:
           limits:
             memory: 200Mi
             cpu: 1
           requests:
             memory: 50Mi
             cpu: 50m
        volumeMounts:
        - name: dataroot
          mountPath: /usr/local/apache2/htdocs
      volumes:
      - name: dataroot
        emptyDir: {}
```

**Note**

Feel free to change the number of replicas (within reason) and the text it is shown in home page of each server, if so desired.

#### 8.3.2 Create the Deployment and interact with the servers using the kubectl tool

- Launch the deployment

```
$ kubectl apply -f test-http.yaml
```

- Re-launch the test-pod from “basic” directory (Refer to Section 6.1) to interact with the web server using local network connection
- Check the pods you have, alongside the IPs they were assigned to:

```
$ kubectl get pods -o wide
```

- Re-create test-pod (if you have deleted it in Section 6.1.2) and log into it

```
$ kubectl apply -f test-pod.yaml
$ kubectl exec -it test-pod -- /bin/sh
```

- Now try to pull the home pages from the two Web servers; use the IPs you obtained above for each of the pod:

```
$ curl http://IP_of_Pod1
$ curl http://IP_of_Pod2
```

  You should get a different answer from the two

### 8.3.3 Load Balancing with Service

Having to manually switch between the two Pods is obviously tedious. What we really want is to have a single logical address that will automatically load-balance between them.
From where you are in the git repository directory,

```
$ cd "your-path-of-repository"/basics
```

Open “test-service.yaml” and examine the code in the file:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    k8s-app: test-svc
  name: test-svc
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    k8s-app: test-http
  type: ClusterIP
```

### 8.3.4 Create the Service and interact with it using the kubectl tool

- Launch the service

```
$ kubectl apply -f test-service.yaml
```

- Look up your service, and write down the IP it is reporting under:

```
$ kubectl get services
```

- Log into the **test-pod** you re-created in the previous section (create a new one if you have deleted it)

```
$ kubectl exec -it test-pod -- /bin/sh
```

- Now from the **test-pod** command line, try to pull the home page from the service IP:

```
$ curl http://IP_of_Service
```

Try it a few times... Which Web server is serving you?

**Note**

You can also use the local DNS name for this (from test-pod)

```
$ curl http://test-svc.<YOUR-NameSpace>.svc.cluster.local
```

## 8.4 Exposing Public Services with Ingress

Sometimes you have the opposite problem; you want to export resources of a single node to the public internet. The above Web services only serve traffic on the private IP network (LAN). If you try curl from your laptops, you will never reach those Pods!
From where you are in the git repository directory:

```
$ cd "your-path-of-repository"/basics
```

Open "test-ingress.yaml" and examine the code in the file:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: traefik
    traefik.ingress.kubernetes.io/router.tls: ""
  name: test-ingress
spec:
  rules:
  - host: test-service.nautilus.optiputer.net
    http:
      paths:
      - backend:
          serviceName: test-svc
          servicePort: 80
        path: /
```

Modify the host name by changing "test-service" to something unique, e.g. your "namespace" name. And then save the file.

### 8.4.1 Create the Ingress and interact with it using the kubectl tool

- Launch the ingress service

  ```
  $ kubectl apply -f test-ingress.yaml
  ```

  You should now be able to fetch the Web pages from your browser by opening "https://**Your-Unique-Name**.nautilus.optiputer.net".

  **Note**

  Nautilus does not provide public IPs to access your services, so the only way to access you web pages is through the host name defined in the .yaml file.

**To Submit**

You need to take a **screen shot** of the result web URL with the web page content for your lab submission.

- You can now delete the deployment:

```
$ kubectl delete -f test-ingress.yaml
```

### 8.5 Clean up your resources

When you are done with the above experimenting, you should clean up any resources that you created for this section using the “kubectl delete” commands that we have learned from above.

**Warning**

DO NOT leave any resources behind as you will forget about them and your namespace could get suspended!

## 9 Example Stateless Application - Deploying PHP Guestbook application with MongoDB

This section shows you how to build and deploy a simple, multi-tier web application using Kubernetes and Docker (*https://kubernetes.io/docs/tutorials/stateless-application/guestbook/*). This example consists of the following components:

- A single-instance MongoDB to store guestbook entries
- Multiple web frontend instances

Steps for this section include the following:

- Start up the backend Mongo database.
- Start up the guestbook frontend web portal.
- Expose and view the frontend service.
- Clean up

From where you are in the git repository directory:

```
$ cd "your-path-of-repository"/guestbook-app
```

### 9.1 Create the Mongo Database Deployment

#### 9.1.1 Open “mongo-deployment.yaml” and examine the code in the file:

-
  ```
  apiVersion: apps/v1
  kind: Deployment
  metadata:
    name: mongo
    labels:
      app.kubernetes.io/name: mongo
      app.kubernetes.io/component: backend
  spec:
  ```

```
  selector:
    matchLabels:
      app.kubernetes.io/name: mongo
      app.kubernetes.io/component: backend
  replicas: 1
  template:
    metadata:
      labels:
        app.kubernetes.io/name: mongo
        app.kubernetes.io/component: backend
    spec:
      containers:
      - name: mongo
        image: mongo:4.2
        args:
          - --bind_ip
          - 0.0.0.0
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
        - containerPort: 27017
```

#### 9.1.2 Create the database Deployment and interact with it using the kubectl tool

- Create the MongoDB Deployment:
  ```
  $ kubectl apply -f mongo-deployment.yaml
  ```
- Query the list of Pods to verify that the MongoDB Pod is running:
  ```
  $ kubectl get pods
  ```
- The response should be similar to this:
  ```
  NAME                            READY     STATUS    RESTARTS   AGE
  mongo-5cfd459dd4-lrcjb          1/1       Running   0          28s
  ```
- Run the following command to view the logs from the MongoDB Deployment:
  ```
  $ kubectl logs -f deployment/mongo
  ```

## 9.2 Create the Mongo Service

The guestbook application needs to communicate to the MongoDB to write its data. You need to apply a Service to proxy the traffic to the MongoDB Pod. A Service defines a policy to access the Pods.

#### 9.2.1 Open "mongo-service.yaml" and examine the code in the file:

-
  ```
  apiVersion: v1
  kind: Service
  metadata:
    name: mongo
    labels:
      app.kubernetes.io/name: mongo
      app.kubernetes.io/component: backend
  spec:
    ports:
    - port: 27017
      targetPort: 27017
    selector:
  ```

```
    app.kubernetes.io/name: mongo
    app.kubernetes.io/component: backend
```

#### 9.2.2 Create the database Service and interact with it using the kubectl tool

- Create the MongoDB Service from the .yaml file
  ```
  $ kubectl apply -f mongo-service.yaml
  ```
- Query the list of Services to verify that the MongoDB Service is running:
  ```
  $ kubectl get service
  ```
  The response should be similar to this:
  ```
  NAME           TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)     AGE
  kubernetes     ClusterIP   10.0.0.1     <none>        443/TCP     1m
  mongo          ClusterIP   10.0.0.151   <none>        27017/TCP    8s
  ```

## 9.3 Create the Guestbook Frontend Deployment

The guestbook application has a web frontend serving the HTTP requests written in PHP. It is configured to connect to the mongo Service to store Guestbook entries.

#### 9.3.1 Open "frontend-deployment.yaml" and examine the code in the file:

-
  ```
  apiVersion: apps/v1
  kind: Deployment
  metadata:
    name: frontend
    labels:
      app.kubernetes.io/name: guestbook
      app.kubernetes.io/component: frontend
  spec:
    selector:
      matchLabels:
        app.kubernetes.io/name: guestbook
        app.kubernetes.io/component: frontend
    replicas: 1
    template:
      metadata:
        labels:
          app.kubernetes.io/name: guestbook
          app.kubernetes.io/component: frontend
      spec:
        containers:
        - name: guestbook
          image: paulczar/gb-frontend:v5
          resources:
            requests:
              cpu: 300m
              memory: 300Mi
            limits:
              cpu: 1
              memory: 500Mi
          env:
          - name: GET_HOSTS_FROM
            value: dns
          ports:
          - containerPort: 80
  ```

#### 9.3.2 Create the frontend Deployment and interact with it using the kubectl tool

- Apply the frontend Deployment from the frontend-deployment.yaml file:
  ```
  $ kubectl apply -f frontend-deployment.yaml
  ```
- Query the list of Pods to verify that the three frontend replicas are running:
  ```
  $ kubectl get pods -l app.kubernetes.io/name=guestbook -l
      app.kubernetes.io/component=frontend
  ```
  The response should be similar to this:
  ```
  NAME                          READY     STATUS    RESTARTS   AGE
  frontend-3823415956-dsvc5     1/1       Running   0          54s
  ```

## 9.4 Create the frontend Service

The mongo Services you applied is only accessible within the Kubernetes cluster because the default type for a Service is ClusterIP. ClusterIP provides a single IP address for the set of Pods the Service is pointing to. This IP address is accessible only within the cluster.

If you want guests to be able to access your guestbook, you must configure the frontend Service to be externally visible, so a client can request the Service from outside the Kubernetes cluster. However a Kubernetes user you can use kubectl port-forward to access the service even though it uses a ClusterIP.

Note: Some cloud providers, like Google Compute Engine or Google Kubernetes Engine, support external load balancers. If your cloud provider supports load balancers and you want to use it, uncomment type: LoadBalancer.

#### 9.4.1 Open "frontend-service.yaml" and examine the code in the file:

-
  ```
  apiVersion: v1
  kind: Service
  metadata:
    name: frontend
    labels:
      app.kubernetes.io/name: guestbook
      app.kubernetes.io/component: frontend
  spec:
    ports:
    - port: 80
    selector:
      app.kubernetes.io/name: guestbook
      app.kubernetes.io/component: frontend
  ```

#### 9.4.2 Create the frontend Service and interact with it using the kubectl tool

- Apply the frontend Service from the frontend-service.yaml file:
  ```
  $ kubectl apply -f frontend-service.yaml
  ```
- Query the list of Services to verify that the frontend Service is running:
  ```
  $ kubectl get services
  ```
  The response should be similar to this:
  ```
  NAME           TYPE         CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
  frontend       ClusterIP    10.0.0.112    <none>        80/TCP         6s
  kubernetes     ClusterIP    10.0.0.1      <none>        443/TCP        4m
  mongo          ClusterIP    10.0.0.151    <none>        6379/TCP       2m
  ```

## 9.5 Exposing the Frontend Service with Ingress

Now we are going to use Ingress rules to expose the frontend service and make it accessible over the public internet.

**Note**

Nautilus does not provide public IPs, so we will only be able to access the web app through dynamic DNS services that Nautilus offers.

### 9.5.1 Open "frontend-ingress.yaml" and examine the code in the file:

-
  ```
  apiVersion: extensions/v1beta1
  kind: Ingress
  metadata:
    annotations:
      kubernetes.io/ingress.class: traefik
      traefik.ingress.kubernetes.io/router.tls: ""
    name: test-ingress
  spec:
    rules:
    - host: test-service.nautilus.optiputer.net
      http:
        paths:
        - backend:
            serviceName: test-svc
            servicePort: 80
          path: /
  ```

### 9.5.2 Create the frontend Ingress and interact with it using the kubectl tool

- Apply the frontend Ingress from the frontend-ingress.yaml file:

  ```
  $ kubectl apply -f frontend-ingress.yaml
  ```

- Now you should now be able to fetch the Web pages from your browser by opening *https://guestbook.nautilus.optiputer.net*, as shown below:

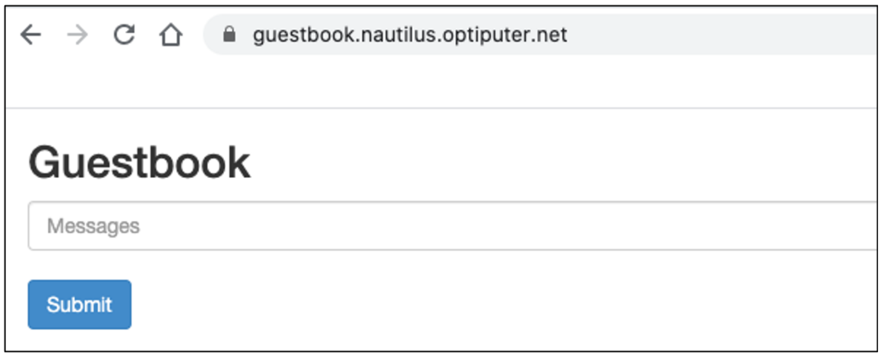


Figure 1: Guestbook web application.

- From the home page, you can add messages into the guest book, which will be displayed on the the same page right away.

**To Submit**

You need to enter some dummy message and then take a **screen shot** of the result page for your lab submission

## 9.6 Verify the stateless nature of your application

### 9.6.1 Delete your application

```
$ kubectl delete -f frontend-service.yaml
$ kubectl delete -f frontend-deployment.yaml
$ kubectl delete -f mongo-service.yaml
$ kubectl delete -f mongo-deployment.yaml
$ kubectl delete -f frontend-ingress.yaml
```

### 9.6.2 Re-create your application

Re-create the entire application by following Section 7.1-7.5.

### 9.6.3 Verify data storage

Open up the link to the home page of the Guestbook application, and check to see if the guest entries that you have entered before are still there.

### 9.6.4 Remove your application and release resources

When you are finished experimenting, you should clean up the resources that you created for it.

In this section, you have learned how to create two pods for a stateless application, Guestbook web application, using deployment, expose the pods using services, and finally using ingress rules to route the services to the internet.

## 10 Example Stateful Application with Deep Learning Pipeline

In this section, we will run an example stateful application – an image analytics application that performs the objects motion detection and classification from a video. This application includes two components, the client and the server. The client streams video and sends video frames to the server, whereas the server takes the frames and performs image analytics. The two components are implemented into microservices encapsulated with Docker containers, communicating via RESTful APIs using the python Flask package. The overall video analytics pipeline is shown in Figure 2 below.

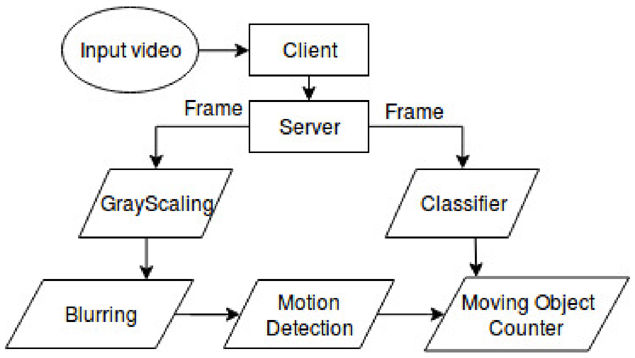


Figure 2: Image processing pipeline to investigate computation offloading.

In this lab, we will deploy this image processing application into two pods on Nautilus Kubernetes cluster, perform example video streaming and objection classification, and collect analytics results in the form of category/number of objects. You will use Deployment to create pods, and Service/Ingress to expose RESTful APIs so Client can communicate with the Server, and finally use Persistent Volumes to store and collect result data. We have prepared Docker container images of both the Client and Server, and you can use them directly in the cluster deployment.

## 10.1 Prepare a Persistent Volume

For this section of the lab, you will need to create a Persistent Volume for your namespace, so you can use the Volume to store your data permanently until you manually delete the data or the Volume.
From where you are in the git repository directory:

```
$ cd "your-path-of-repository"/pvc
```

### 10.1.1 Open "pvc.yaml" and examine the code in the file:

-
  ```
  apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    name: mizzoucerivolume
  spec:
    storageClassName: rook-cephfs-east
    accessModes:
    - ReadWriteMany
    resources:
      requests:
        storage: 200Mi
  ```

  **Warning**

  You need to change the name of PVC ("mizzoucerivolume") to something unique to you. Once you change the PVC name here, you will also need to update the name in your Server and Client manifest files to match to the name you use here.

### 10.1.2 Create a Persistent Volume for your Namespace

-
  ```
  $ kubectl apply -f pvc.yaml
  ```

- Check the status of the Persistent Volume

  ```
  $ kubectl get pvc
  ```

  You should see the "STATUS" as "Bound", which means your requested Persistent Volume is ready to use to persist your data.

  ```
  NAME      STATUS    VOLUME              CAPACITY ACCESS MODES   STORAGECLASS        AGE
  volume    Bound     pvc-6a054b00-...     200Mi       RWX        rook-cephfs-east     1d
  ```

## 10.2 Create the Server Node to Accept Streamed Video Frames

Now we are going to start the Server node of the application to listen to streamed video frames from any Client nodes.
From where you are in the git repository directory:

-
  ```
  $ cd "your-path-of-repository"/ImageProcessingWebServices
  ```

### 10.2.1 Create the Server Service with Attached Ingress Policy

In this part, we show you how we can just use one yaml file to configure the Deployment, Service and Ingress altogether for our application. By doing this, you do not need to create separate configuration files, and create the resources separately. The idea of this is to use "− − −" to include and separate different documents in the same yaml file. In our below yaml file, we have three documents, each defining a resource (i.e. Deployment, Service and Ingress) in our Kubernetes cluster.

### 10.2.2 Open "server-deployment.yaml" and examine the code in the file:

-
  ```
  apiVersion: apps/v1
  kind: Deployment
  metadata:
    name: image-processing
    labels:
      app: imageProcessing
      tier: server
  spec:
    replicas: 1
    selector:
      matchLabels:
        app: imageProcessing
        tier: server
    template:
      metadata:
        labels:
          app: imageProcessing
          tier: server
      spec:
        containers:
        - name: imgproc-server
          image: wangso/imgproc-server:V1
          imagePullPolicy: Always
          resources:
            limits:
              memory: 2000Mi
              cpu: 2
            requests:
              memory: 1000Mi
              cpu: 1
          ports:
            - containerPort: 5000
          volumeMounts:
          - mountPath: /ImageProcessingWebServices/output/server
            name: mizzoucerivolume
        volumes:
        - name: mizzoucerivolume
          persistentVolumeClaim:
            claimName: mizzoucerivolume
        restartPolicy: Always
  ---
  apiVersion: v1
  kind: Service
  metadata:
  ```

```
  name: server-service
spec:
  type: ClusterIP
  selector:
    app: imageProcessing
    tier: server
  ports:
  - name: http
    port: 5000
    targetPort: 5000
    protocol: TCP
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: traefik
    traefik.ingress.kubernetes.io/router.tls: ""
  name: server-ingress
spec:
  rules:
  - host: wangso-server.nautilus.optiputer.net
    http:
      paths:
      - backend:
          serviceName: server-service
          servicePort: 5000
        path: /
```

**Warning**

You need to change the ”host:” field in the Ingress portion of the manifest file to include your own unique hostname to avoid conflict with other students, e.g. you can use ”Your-Namespace.nautilus.optioputer.net”.

**Warning**

You need to update the “claimName: mizzoucerivolume” field in the image-processing Deployment part of the manifest file above to match your PVC claimName defined in your pvc.yaml.

- Create the Server node from the yaml file:

```
$ kubectl apply -f server-deployment.yaml
```

- You should expect to see the following output:

```
deployment.apps/image-processing created
service/server-service created
ingress.extensions/server-ingress created
```

- Query the list of Pods to verify that the Server Pod is running:

```
$ kubectl get pods
```

- The response should be similar to this right after you create the deployment:

```
NAME                                READY   STATUS              RESTARTS   AGE
image-processing-56b6cdf898-x7hhm   0/1     ContainerCreating   0          77s
```

It will take several minutes for the pod to download the Docker image, then you will see that the pod is running:

```
NAME                              READY   STATUS    RESTARTS   AGE
image-processing-56b6cdf898-x7hhm 1/1     Running   0          2m5s
```

- Check the deployment

  ```
  $ kubectl get deployments
  ```

  You will see the output:

  ```
  NAME               READY   UP-TO-DATE   AVAILABLE   AGE
  image-processing   1/1     1            1           3m54s
  ```

- Run the following command to view the logs from the Server Deployment:

  ```
  $ kubectl logs -f deployment/image-processing
  ```

  You will see output similar to the following, which indicates that Server is up and running:

  ```
   * Serving Flask app "server" (lazy loading)
   * Environment: production
     WARNING: This is a development server. Do not use it in a production
      deployment.
     Use a production WSGI server instead.
   * Debug mode: on
   * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
   * Restarting with stat
   * Debugger is active!
   * Debugger PIN: 365-795-907
  ```

### 10.2.3 Update the Server Address on the Server

**Warning**

This step is critical! If you do not update the Server address, you won't be able to communicate between your Server and Client machines, and the video streaming and analytics pipeline will NOT work!

- To double check your Server address:

  ```
  $ kubectl get ingress
  ```

  You should see the output like this:

  ```
  NAME             CLASS    HOSTS                                ADDRESS   PORTS
     AGE
  server-ingress   <none>   wangso-server.nautilus.optiputer.net             80
     1h
  ```

  And the HOSTS field is what your Server's address should be at.

- From your local computer, keep the Server terminal up, open up another terminal, and run the following command (Replace both of the "Your_Server_Ingress_Address" with the Ingress HOST name you specified in previous section (Section 8.2.2)) (keep the double quotes in the -d argument):

  ```
  $ curl -X POST -H 'Content-Type: application/json'
     https://Your_Server_Ingress_Address/setNextServer -d
     '{"server":"Your_Server_Ingress_Address"}'
  ```

  **Warning**

  Please DO NOT copy the command from this pdf instruction, as some of the symbols will be changed upon copy/paste. The command is available for you to edit/copy in the commands.txt file located in the same directory.

- After running the command, you will see a chunk of standard output from the Server terminal with some error message. Leave this terminal running and proceed to the next section in a new terminal.

## 10.3 Create the Client Node to Stream Video to the Server

The Client side will be used to send video streams to the Server for image processing and object recognition/counting. For the Client, since we are not exposing its address, we will just need a deployment document in the manifest file.

### 10.3.1 Open "client-deployment.yaml" and examine the code in the file:

-
  ```
  apiVersion: apps/v1
  kind: Deployment
  metadata:
      name: imgproc-client
  spec:
      replicas: 1
      selector:
          matchLabels:
              app: imageProcessing
              tier: client
      template:
          metadata:
              labels:
                  app: imageProcessing
                  tier: client
          spec:
              containers:
              - name: imgproc-client
                image: wangso/imgproc-client:V1
                imagePullPolicy: Always
                env:
                  - name: server
                    value: "Your_Server_Ingress_Name"
                resources:
                  limits:
                    memory: 2000Mi
                    cpu: 2
                  requests:
                    memory: 1000Mi
                    cpu: 1
                volumeMounts:
                - mountPath: /ImageProcessingWebServices/output/client
                  name: mizzoucerivolume
              volumes:
              - name: mizzoucerivolume
                persistentVolumeClaim:
                  claimName: mizzoucerivolume
              restartPolicy: Always
  ```

**Warning**

You will need to update the "value: Your_Server_Ingress_Name" field in the manifest file to match the Ingress host name of your Server (see Section 8.2.2). If it does not match to your Server address, your Client will not be able to connect to your Server!

**Warning**

If you modified the Persistent Volume in the pvc.yaml file, you will need to update the ”claimName:” field in the manifest file above to match your PVC claimName.

- Create the Client Deployment

```
$ kubectl apply -f client-deployment.yaml
```

  You will see that deployment has been created:

```
deployment.apps/imgproc-client created
```

- Query the list of Pods to verify that the Client is running:

```
$ kubectl get pods
```

  The response should be similar to this:

```
NAME                                 READY    STATUS     RESTARTS    AGE
image-processing-56b6cdf898-x7hhm    1/1      Running    0           45m
imgproc-client-5fd5d87556-5xr45      1/1      Running    0           59s
```

- On your Server terminal, you will see output messages that indicate your application is running – Client is sending video frames to the Server and the Server is performing object recognition and counting. Let it run for a while until you see something similar to the following:

```
Initializing Application
Classifying
<type 'cv2.dnn_Net'>
detecting
detecting
detecting
('car', 1, 0.9873948693275452)
('car', 2, 0.9790951013565063)
('car', 3, 0.9716188311576843)
('car', 4, 0.9575230479240417)
('bus', 1, 0.8878790736198425)
('car', 5, 0.8661065101623535)
('truck', 1, 0.8458478450775146)
('car', 6, 0.786876916885376)
('car', 7, 0.7390177249908447)
('truck', 2, 0.5835233330726624)
('car', 8, 0.5516889691352844)
('car', 9, 0.5080972909927368)
```

## 10.4 Log into either Server or Client pod to see the output result data

**Note**

The output data will be in the ../output/server directory if you login to the Server Pod, and in the ../output/client directory if you login to the Client pod.

- Log into your pod:

```
$ kubectl exec -it Your_Pod_Name bash
```

  e.g. $ *kubectl exec -it image-processing-56b6cdf898-x7hhm bash*

- Then go to the output directory:

```
$ cd ../output/server
```

- You will see the output result files:

```
$ ls -la
total 5
drwxrwxrwx 1 root root    2 Mar 26 21:14 .
drwxr-xr-x 4 root root   34 Mar 23 04:53 ..
-rw-r--r-- 1 root root 1109 Mar 26 21:18 output.txt
-rw-r--r-- 1 root root 3198 Mar 26 21:18 result.txt
```

- The output.txt will be the object classification results. And the result.txt is the performance data.

  E.g. To show the content of the output.txt file:

```
$ cat output.txt
  {Your results ...}
```

In the output.txt, you can see the list of our objects and the number of occurrence of each object in the streamed video.

**To Submit**

You need to take a **screen shot** of the output result for your lab submission.

Since our example video is traffic monitor, our result will mostly contain object counts for vehicles, such as cars, trucks, etc. These result show a sum of the count of the main object in each of the video frame, which is shown below in Figure 3.

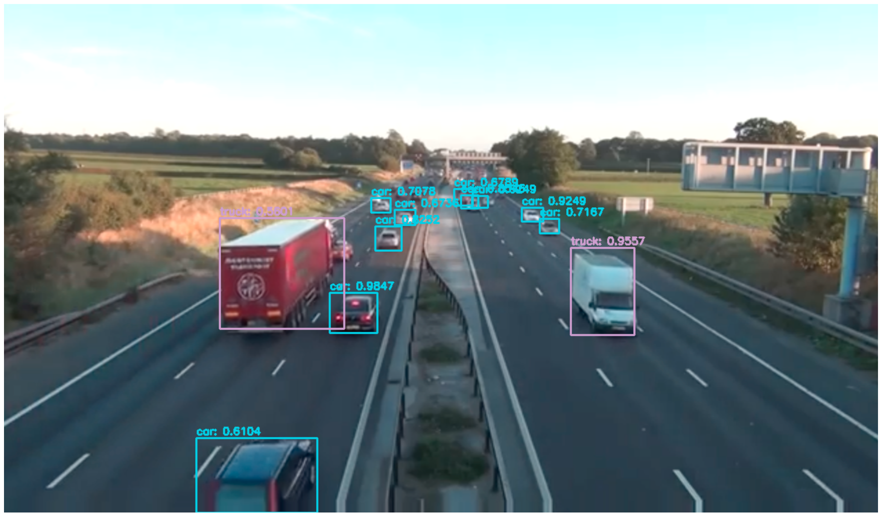


Figure 3: Example of the moving object classification results on the ground surveillance video from the drone.

To show the content of the result.txt file:

```
$ cat result.txt
    {Your results ...}
```

**To Submit**

You need to take a **screen shot** of the output result for your lab submission.

In the result.txt, the “frame” is the number of frame in the original streamed video file, and “fps” shows the speed of processing each of the frames in seconds.

**Note**

In the yaml file, you might have noticed that we used Persistent Volume we have claimed (by using PersistentVolumeClaim) and mounted on both the Server and Client. This means that the output data are persisted on the mounted volume and will be accessible permanently unless you delete them purposely. In this case, even if your pod dies/restarts for some reason, or you have deleted your deployment, the data will not get lost, and you will still be available to access them.

#### 10.4.1 Delete your resources

Remove the resources that you created for this section.

```
$ kubectl delete -f server-deployment.yaml
$ kubectl delete -f client-deployment.yaml
```

## 11 Homework Problems

1) Describe Pod, Deployment, Service, Ingress, Persistent Volumes, and what they are used for.
2) Describe the difference among ClusterIP, NodePort and Load Balancing, and what they are best suited for.
3) Describe the difference among port, target port and node port, and what they are best used for.
4) Describe the difference between Ingress and port-forwarding, and what they are best used for.

## 12 Lab Report

1) Submit the required screen shots in the lab steps
2) Short essay answers to the homework problems