

CO3219 Internet and Cloud Computing

Coursework: Cloud System Design and Evaluation

Name: Thomas Cooper

ID: tcc16

Group Number: 13

Other Group Members: Worked alone (MC)

1 Design and Evaluation of the Private Cloud

As identified from IBM, cloud computing can offer many benefits. These include flexibility, efficiency, and strategic value [1]. With these benefits in mind, it will be useful to examine and compare existing cloud technologies.

Three major existing technologies are Amazon Web Services (AWS), Microsoft Azure, and Google Cloud. There are also other open-source providers such as OpenStack. Choosing AWS would mean access to their data centres, which are “thought to be the largest of all cloud service providers” [2]. This would mean any downtime would be virtually nonexistence. Complaints with AWS include a complicated fee and feature structure which may mean heavy research and investigation for the user. Azure has the advantage of being a Microsoft provided cloud. This means that there is high integration with Windows OS's and other Microsoft systems. Negatives associated with Azure is that they do not offer very many features and they are not as developed as other technologies. Google Cloud has the advantage of being a cutting-edge leader in AI and how they incorporate AI into their cloud services. They do, however, not offer the large amount of data centres as AWS or Azure. Open-source services are a final choice, with OpenStack being the most popular. The obvious advantage is that that are free to use, however may lack the robust documentation or community to assist with support.

I will be setting up a private cloud to enable the use of a whiteboard style application on a browser. It will be able to synchronise across multiple instances.

The first choice would be how to create the application. Docker was chosen for this. The whiteboard application itself was taken from the example provided on Blackboard. With this application an image can be created easily in Docker.

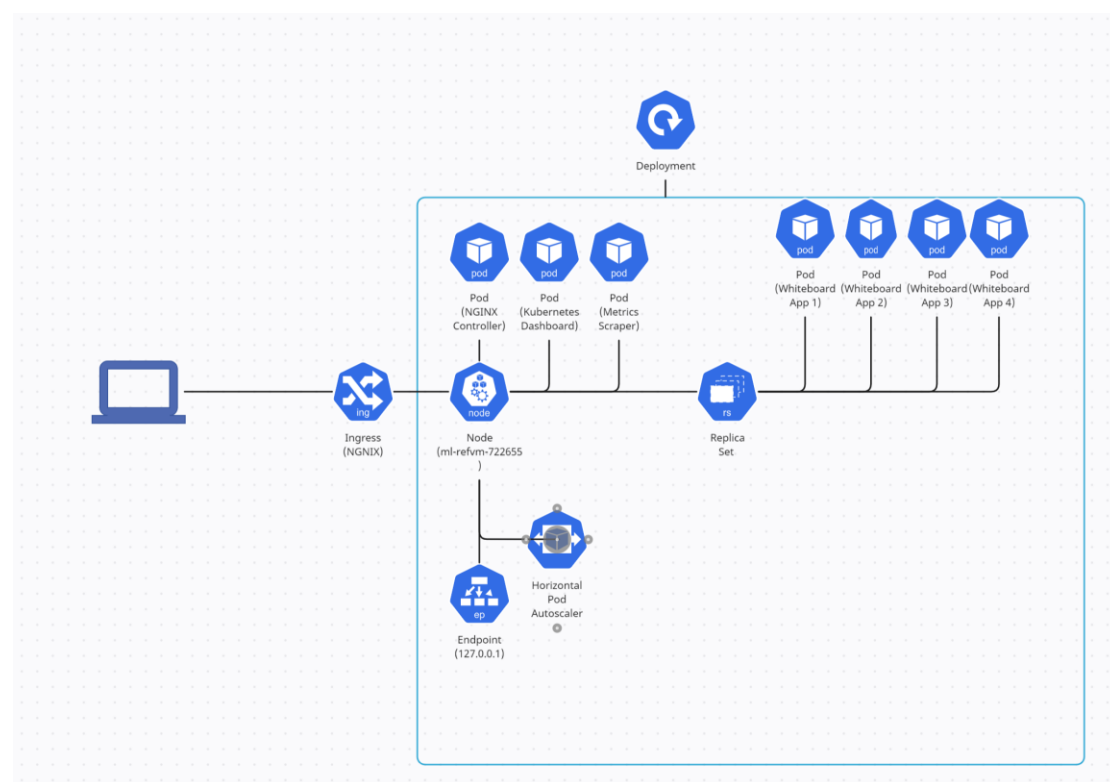
As it is a private cloud that will be created, I have decided to use Kubernetes. It is a container manager and can run Docker images and the contained applications. One of the main reasons for this is the automated operations that the platform can provide, including scalability. Another reason for the choice of Kubernetes is that there is high a amount of abstraction for the user and developer. This means that most of the underlying environment is set up without the need for user intervention or initialization. Essentially, it makes the platform simpler to use. Kubernetes also has features where health of the running containers is monitored. One last advantage is that Kubernetes provides a GUI for managing the platform.

With Kubernetes, the architect is based off nodes and clusters (a cluster being the overall working platform). The nodes are the working machines. In my case I plan to run one main master node called “ml-refvm-722655”. In this node there are many pods

running, mostly for control and monitoring. A pod in this case is the running containers that the node is providing. Most pods are background pods to run applications such as the dashboard and metric scraper to monitor the health of the system (which assists with consistency). An ingress system (from NGINX) with built in load balancing will also be used [3]. I will initialise the Kubernetes deployment with 4 pods (4 replicas to guarantee availability if auto scaling is defective) and add built in scaling features. The choice for this will be a horizontal pod autoscaler, where pods will be scaled up or down as required [4]. Latency issues will not be much of an issue as the application will be running locally. The only problem that may occur is if too many requests attempt a connection at the same time. In Kubernetes, working pods are always monitored and therefore a high level of consistency is provided as they are restarted as required and can be assigned to incoming requests from the ingress controller. For example, if a pod is inactive and a request from the ingress controller comes in, it will be sent to the pod that is inactive.

Figure 1 shows the design I plan to implement.

Figure 1



2 Implementation of the Private Cloud

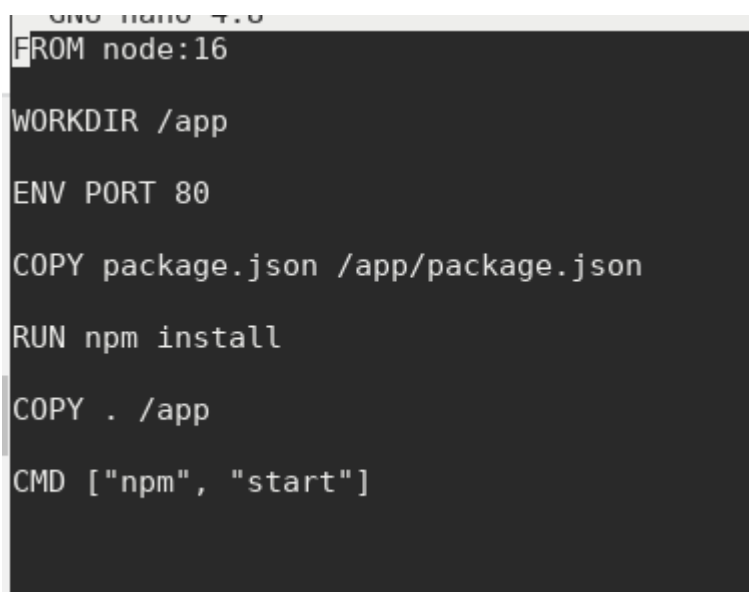
The first take in the implementation of the private cloud is to build the container that will be run. This image is created through Docker, and this means installing Docker on the machine. A combination of simple commands achieves this as seen in *figure 2*.

Figure 2

1. `sudo apt update`
2. `sudo apt install apt-transport-https ca-certificates curl gnupg-agent software-properties-common`
3. `curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -`
4. `sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"`
5. `sudo apt-get update`
6. `sudo apt-get install docker-ce docker-ce-cli containerd.io`

With Docker installed it is now time to create my container with the whiteboard application. As the whiteboard application was the example from Blackboard, after examining the files I knew I would need a NodeJS to run and a software package installer, in this case NPM. With all this information I can create a Dockerfile. This is a file which contains all the instructions Docker needs to build an image [5]. The Dockerfile will need to be in the same directory as the whiteboard application. *Figure 3* shows my Dockerfile.

Figure 3



```
FROM node:16
WORKDIR /app
ENV PORT 80
COPY package.json /app/package.json
RUN npm install
COPY . /app
CMD ["npm", "start"]
```

As seen in *figure 3*, the Dockerfile uses NodeJS in the directory of “/app”. It then copies

all the json files required for the whiteboard application and uses NPM to start the application, therefore creating the image. In *figure 4* using a command to view all docker images, we can see the creation of the image was successful.

Figure 4

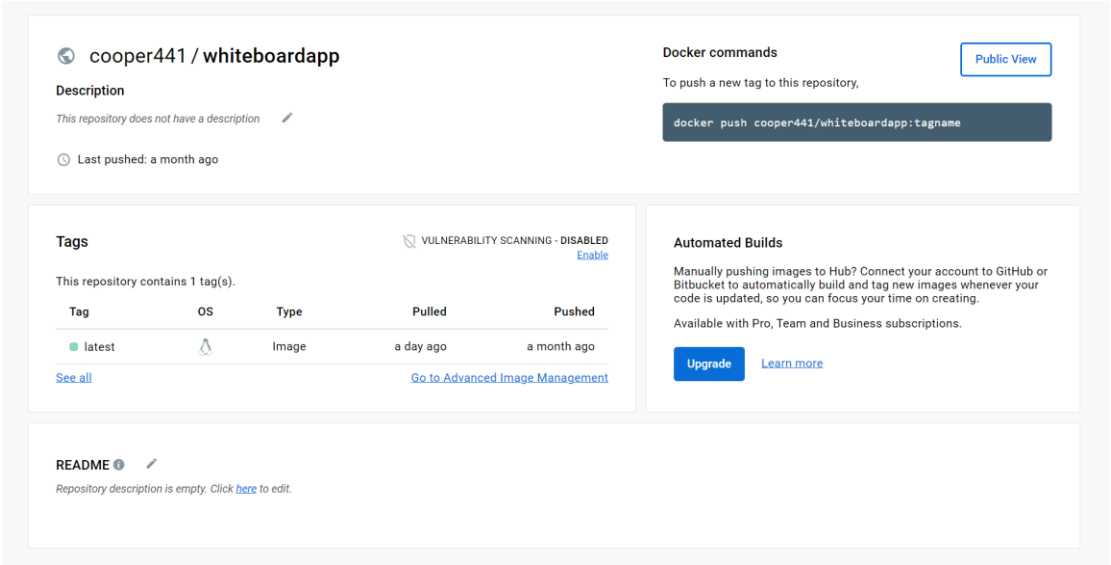
```
uolcoadmin@ML-RefVm-722655:~/whiteboard$ sudo docker images
REPOSITORY          TAG         IMAGE ID      CREATED       SIZE
cooper441/whiteboardapp latest      4bac3e012f1d  3 weeks ago  925MB
cooper441/whiteboardapp public      4bac3e012f1d  3 weeks ago  925MB
```

For Kubernetes to access this image, however, it is useful to be available online. For the reason, I created a Docker Hub account and pushed this image to an online repository. Although not strictly necessary, if I were to update the image and application, I could push the updates to the Docker Repo quite simply and update labels. Once a Docker Hub was created and signed in, I could then issue the command to push the new image as seen in *figure 5*. *Figure 6* shows the online browser of Docker Hub with my image successfully on the repository.

Figure 5

```
uolcoadmin@ML-RefVm-722655:~/whiteboard$ docker push cooper441/whiteboardapp:latest
```

Figure 6



Although not necessary for this exercise, I went ahead and simply pulled the image from the Docker hub. Next, I used Kubernetes to create a deployment. This can be achieved a few ways, including writing a .yaml file (a configuration file for Kubernetes). Instead, I used a simple command to create the deployment as the Docker image was accessible locally (see *figure 7*).

Figure 7

```
uolcoadmin@ML-RefVm-722655:~/whiteboard$ sudo microk8s kubectl create deployment whiteboardapp2 --image=cooper441/whiteboardapp
```

The deployment now needs to be exposed. This is achieved with a simple command in *figure 8*

Figure 8

```
uolcoadmin@ML-RefVm-722655:~/whiteboard$ sudo microk8s kubectl expose whiteboardapp2
```

After exposing the deployment a command in can be used as seen *figure 9 and 10* to check everything is working so far.

Figure 9

```
uolcoadmin@ML-RefVm-722655:~$ sudo microk8s kubectl get all --all-namespaces
```

Figure 10

default	service/whiteboardapp2	ClusterIP	10.152.183.80	<none>	80/TCP	28h
default	deployment.apps/whiteboardapp2	4/4	4	4		28h

The next step is access to the immensely powerful and useful Kubernetes GUI dashboard. This avoids the use of many commands in the terminal. As we can see in *figure 11*, the service Kubernetes-dashboard is accessed through IP 10.152.183.28 on port 443.

Figure 11

kube-system	service/kubernetes-dashboard	ClusterIP	10.152.183.28	<none>	443/TCP	33d
-------------	------------------------------	-----------	---------------	--------	---------	-----

In order to gain access and for security, a token needs to be created and used to log into the dashboard. This is achieved with the simple command in *figure 12*,

Figure 12

```
sudo microk8s kubectl -n kube-system describe secret $token
```

This token provided by the *figure 12* command can now be used to log into the dashboard as seen in *figure 13*. Once logged a landing page can be seen in *figure 14*.

Figure 13

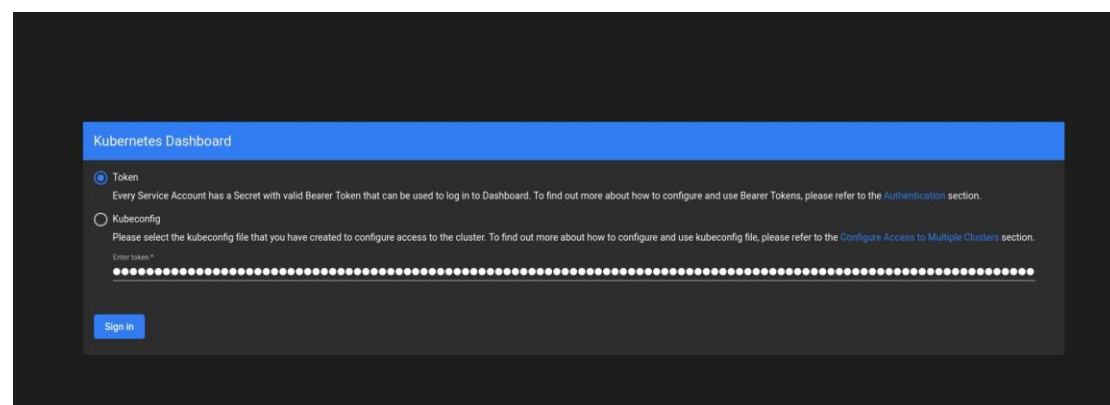
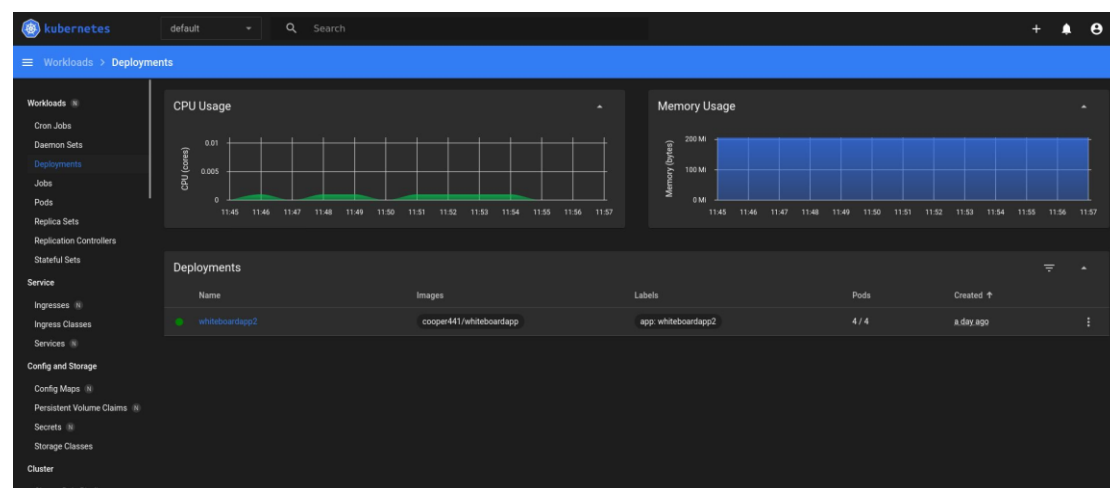


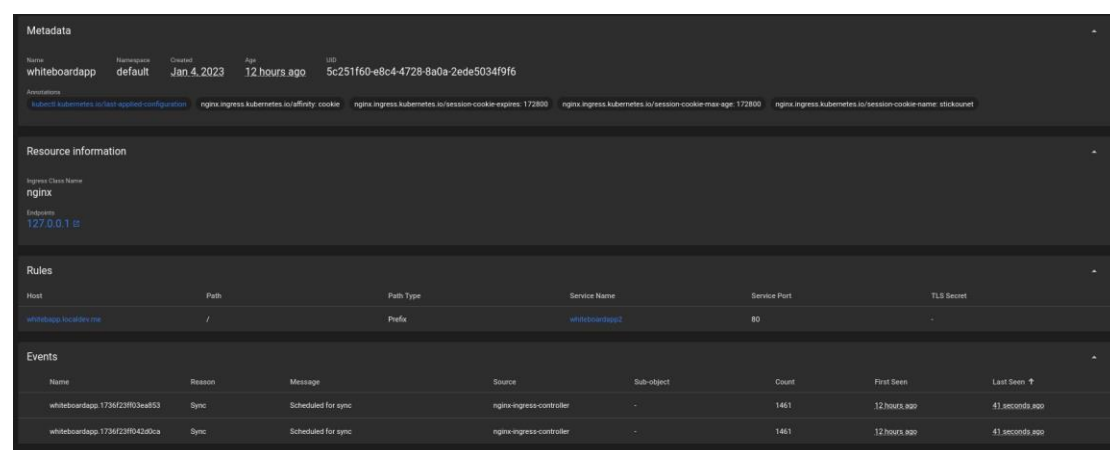
Figure 14



Immediately it is apparent how useful this GUI dashboard is as it shows memory and CPU usage in a graph form. This would allow instant assessment of deployment performance, even with multiple deployments. The .yaml file for the deployment can be accessed on this dashboard and can be seen in *figure 15* in the appendix.

Whilst the whiteboard application can now technically be run through 4 replicas pods created by accessing their individual IP address, this is not very effective. Instead, I decided to implement an ingress controller with a single host name to be used to access a pod containing the application. NGINX was used as it has a built-in load balancer [3]. Essentially, incoming requests will be sent to a pod available not in use first. In figure 16, the information regarding the ingress controller can be seen.

Figure 16



If all pods are being used the auto scaler will come into effect. This is a built-in feature of Kubernetes and was set up with the simple command in *figure 17*

Figure 17

```
root@coolamingMI-RefVm-722655:~/whiteboard$ sudo microk8s kubectl autoscale deployment whiteboardapp2 --min=2 --max=10 cpu-percent=90
```

In figure 17 I am setting the auto scaler on my deployment whiteboardapp2 with a

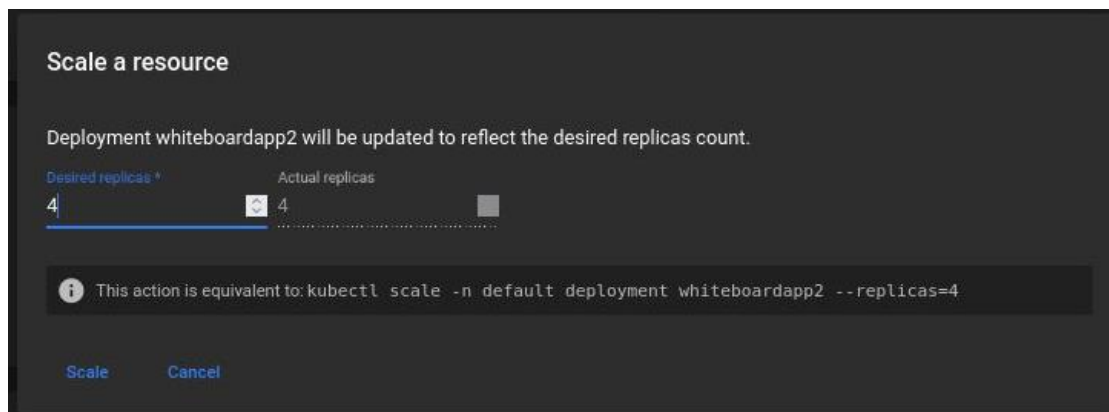
minimum of 2 (although changed to 4 later) pods and a maximum of 10, with using no more than 90% of CPU resource availability. When this was first implemented, however, I was receiving errors relating to the resource settings of the pods themselves. In this case I must edit the .yaml file for the deployment to allow specific resource management on the pods so the auto scaler could work as desired. *Figure 18* shows the edit I had to make, previously the pods had no set resource settings. This information was gathered from both the documentation for resource management [6] and the horizontal pod autoscaling documentation [7].

Figure 18

```
resources:
  limits:
    cpu: '1'
  requests:
    cpu: 100m
```

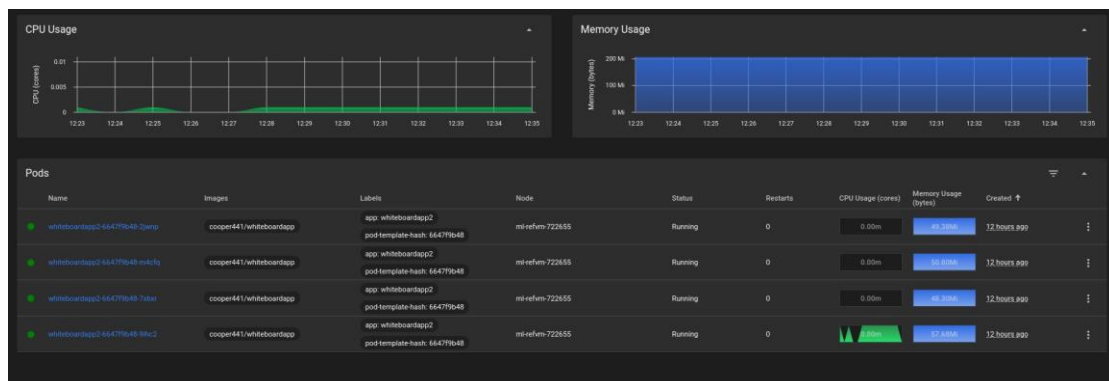
As well as auto scaling, the deployment can be scaled manually through the Kubernetes dashboard. This can be seen in *figure 19*

Figure 19



The current number of pods and their resource usage can be seen in *figure 20*

Figure 20



3 Implementation and Deployment Distributed Whiteboard Application

I have used the whiteboard example provided on Blackboard. As seen in *figures 21 and 22*, there are two tabs open on the browser using the ingress address domain (which reroutes the request to the default local IP address of 127.0.0.1).

Figure 21

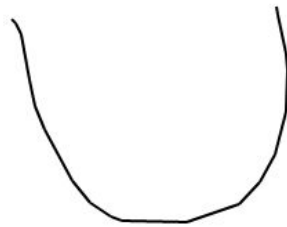
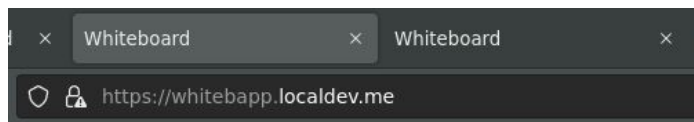
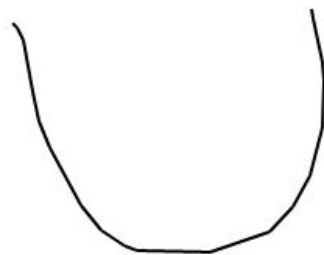
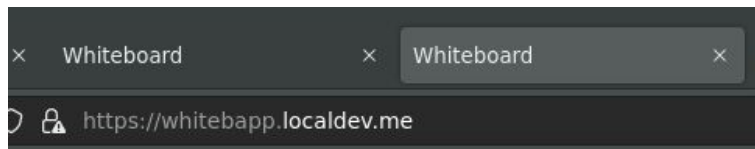


Figure 22



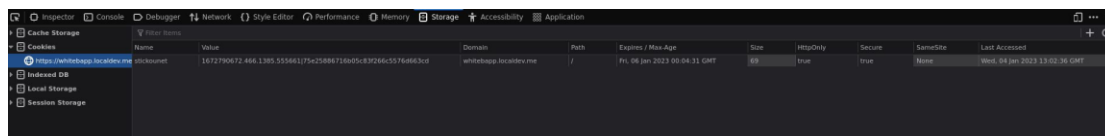
What I have noticed, however, is when a new user joins (i.e., a new tab or window is opened) they do not have access to what has already been written on the whiteboard. However, what they do write will be reflected on the previous users whiteboard. With this said, multiple users can still draw simultaneously on a shared canvas, only that new users joining will not see what has previously been drawn.

To achieve a maintained state across users of the whiteboard application, “sticky sessions” was implemented in the NGINX ingress controller. This is not the most ideal solution as all client requests are sent to the same pod and with further work, a better solution will need to be found. However, figure 23 shows the edit to ingress .yaml file that was made to achieve consistency on one pod. One advantage of using sticky sessions, however, is that through cookies on the browser, the writings on the canvas will be maintained even when the window/tab is closed. These cookies can be seen in figure 24.

Figure 23

```
nginx.ingress.kubernetes.io/affinity: cookie
nginx.ingress.kubernetes.io/session-cookie-expires: '172800'
nginx.ingress.kubernetes.io/session-cookie-max-age: '172800'
nginx.ingress.kubernetes.io/session-cookie-name: stickounet
```

Figure 24



Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Last Accessed
stickounet	1672790672.466.1385.553681175e25086716d05c83266c5576d863cd	whiteboard.localdev.me	/	Fri, 06 Jan 2023 06:04:31 GMT	69	true	true	None	Wed, 04 Jan 2023 13:02:36 GMT

4 Demonstration of the Private Cloud

Individual contribution:

As I have completed this assignment on my own (due to MC), all work was completed by myself. This includes:

1. Install Microk8s
2. Creating Docker image
3. Creating and pushing Docker image to Docker Hub
4. Pulling image from Docker Hub
5. Setting up Kubernetes through Microk8s
6. Creating a deployment on Kubernetes
7. Creating replica pods
8. Implementing NGINX ingress controller with load balancing
9. Testing and debugging
10. Implementing a horizontal auto scaler
11. Implementing cookies and sticky sessions with the ingress controller

Group Demo Video:

<https://youtu.be/Ju2T-dRgzSc>

5 Critical Review of the Private Cloud

My first impression is that Docker was very simple and straight forward to use. Kubernetes on the other hand was a more complex simple, although mitigated somewhat by the powerful GUI dashboard.

I believe the implementation of the Kubernetes system went relatively well, however, there was a lot of troubleshooting and roadblocks which required a lot of time to research and resolve. Once understood more, it is a very powerful system to use.

The whiteboard application itself could be better. I used the example from Blackboard with no additions made. If I could re-do the assignment, I would spend more time making the whiteboard application itself better.

Key challenges I faced were the implementation of the ingress controller. Many attempts were made to get it working and even in the end improvements could be made. For example, the ingress controller only sends to one pod, whereas it should be load balancing more effectively. This is due to the sticky sessions I had to implement to ensure consistency across instances of the application.

The auto scaler, on the other hand, was very simple to implement and provides a solid basis for scalability.

6 References (optional)

[1] IBM Cloud Education, "Benefits of cloud computing," IBM. [Online]. Available: <https://www.ibm.com/uk-en/cloud/learn/benefits-of-cloud-computing>. [Accessed: 22-Dec-2022].

[2] PRO OnCall Technology, "Comparing the 3 major cloud computing platforms," PRO OnCall Technology, 27-May-2020. [Online]. Available: <https://prooncall.com/comparing-the-3-major-cloud-computing-platforms/>. [Accessed: 02-Jan-2023].

[3] "Overview," NGINX Docs. [Online]. Available: <https://docs.nginx.com/nginx-ingress-controller/intro/overview/#:~:text=The%20Ingress%20Controller%20is%20an,cloud%20load%20balancer%20running%20externally>. [Accessed: 03-Jan-2023].

[4] "Horizontal pod autoscaling," Kubernetes, 26-Nov-2022. [Online]. Available: <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>. [Accessed: 03-Jan-2023].

[5] "Dockerfile reference," Docker Documentation, 03-Jan-2023. [Online]. Available: <https://docs.docker.com/engine/reference/builder/#:~:text=A%20Dockerfile%20is%20a%20text,can%20use%20in%20a%20Dockerfile%20>. [Accessed: 04-Jan-2023].

[6] "Resource Management for pods and containers," Kubernetes, 20-Oct-2022. [Online]. Available: <https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#requests-and-limits>. [Accessed: 04-Jan-2023].

[7] "Horizontal pod autoscaling," Kubernetes, 26-Nov-2022. [Online]. Available: <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>. [Accessed: 04-Jan-2023].

Appendix (optional)

Figure 15

```

kind: Deployment
apiVersion: apps/v1
metadata:
  name: whiteboardapp2
  namespace: default
  uid: 2a0d45de-f061-41b4-ba7d-a9c23b9ef865
  resourceVersion: '4158358'
  generation: 9
  creationTimestamp: '2023-01-03T06:55:13Z'
  labels:
    app: whiteboardapp2
  annotations:
    deployment.kubernetes.io/revision: '4'
managedFields:
- manager: kubelite
  operation: Update
  apiVersion: apps/v1
  fieldsType: FieldsV1
  fieldsV1:
    f:spec:
      f:replicas: {}
      subresource: scale
- manager: kubectl-create
  operation: Update
  apiVersion: apps/v1
  time: '2023-01-03T06:55:13Z'
  fieldsType: FieldsV1
  fieldsV1:
    f:metadata:
      f:labels:
        .: {}
      f:app: {}
    f:spec:
      f:progressDeadlineSeconds: {}
      f:revisionHistoryLimit: {}
      f:selector: {}
      f:strategy:
        f:rollingUpdate:
          .: {}
          f:maxSurge: {}
          f:maxUnavailable: {}
        f:type: {}
      f:template:
        f:metadata:
          f:labels:
            .: {}
          f:app: {}
        f:spec:
          f:containers:
            k:{"name":"whiteboardapp"}:
              .: {}
              f:image: {}
              f:imagePullPolicy: {}
              f:name: {}
              f:ports:
                .: {}
                k:{"containerPort":80,"protocol":"TCP"}:
                  .: {}
                  f:containerPort: {}
                  f:protocol: {}
              f:resources: {}
              f:terminationMessagePath: {}
              f:terminationMessagePolicy: {}
            f:dnsPolicy: {}
            f:restartPolicy: {}
            f:schedulerName: {}
            f:securityContext: {}
            f:terminationGracePeriodSeconds: {}
- manager: dashboard
  operation: Update
  apiVersion: apps/v1
  time: '2023-01-03T23:42:17Z'
  fieldsType: FieldsV1
  fieldsV1:
    f:spec:
      f:template:
        f:metadata:
          f:annotations:

```

```

      f:annotations:
        .: {}
        f:kubectl.kubernetes.io/restartedAt: {}
    f:spec:
      f:containers:
        k:{"name":"whiteboardapp"}:
          f:resources:
            f:limits:
              .: {}
              f:cpu: {}
            f:requests:
              .: {}
              f:cpu: {}
- manager: kubelite
  operation: Update
  apiVersion: apps/v1
  time: '2023-01-04T00:23:10Z'
  fieldsType: FieldsV1
  fieldsV1:
    f:metadata:
      f:annotations:
        .: {}
        f:deployment.kubernetes.io/revision: {}
    f:status:
      f:availableReplicas: {}
      f:conditions:
        .: {}
        k:{"type":"Available"}:
          .: {}
          f:lastTransitionTime: {}
          f:lastUpdateTime: {}
          f:message: {}
          f:reason: {}
          f:status: {}
          f:type: {}
        k:{"type":"Progressing"}:
          .: {}
          f:lastTransitionTime: {}
          f:lastUpdateTime: {}
          f:message: {}
          f:reason: {}
          f:status: {}
          f:type: {}
      f:observedGeneration: {}
      f:readyReplicas: {}
      f:replicas: {}
      f:updatedReplicas: {}
      subresource: status
spec:
  replicas: 4
  selector:
    matchLabels:
      app: whiteboardapp2
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: whiteboardapp2
      annotations:
        kubectl.kubernetes.io/restartedAt: '2023-01-03T23:42:17Z'
    spec:
      containers:
        - name: whiteboardapp
          image: cooper441/whiteboardapp
          ports:
            - containerPort: 80
              protocol: TCP
          resources:
            limits:
              cpu: '1'
            requests:
              cpu: 100m
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
          imagePullPolicy: Always
      restartPolicy: Always

```

```
3 restartPolicy: Always
4 terminationGracePeriodSeconds: 30
5 dnsPolicy: ClusterFirst
6 securityContext: {}
7 schedulerName: default-scheduler
8 strategy:
9   type: RollingUpdate
10   rollingUpdate:
11     maxUnavailable: 25%
12     maxSurge: 25%
13   revisionHistoryLimit: 10
14   progressDeadlineSeconds: 600
15 status:
16   observedGeneration: 9
17   replicas: 4
18   updatedReplicas: 4
19   readyReplicas: 4
20   availableReplicas: 4
21   conditions:
22     - type: Progressing
23       status: 'True'
24       lastUpdateTime: '2023-01-03T23:42:25Z'
25       lastTransitionTime: '2023-01-03T06:55:13Z'
26       reason: NewReplicaSetAvailable
27       message: ReplicaSet "whiteboardapp2-6647f9b48" has successfully progressed.
28     - type: Available
29       status: 'True'
30       lastUpdateTime: '2023-01-04T00:23:09Z'
31       lastTransitionTime: '2023-01-04T00:23:09Z'
32       reason: MinimumReplicasAvailable
33       message: Deployment has minimum availability.
```