CO7219 Internet and Cloud Computing

Report: Cloud System Design and Evaluation

Task 1: Private Cloud Design Architecture

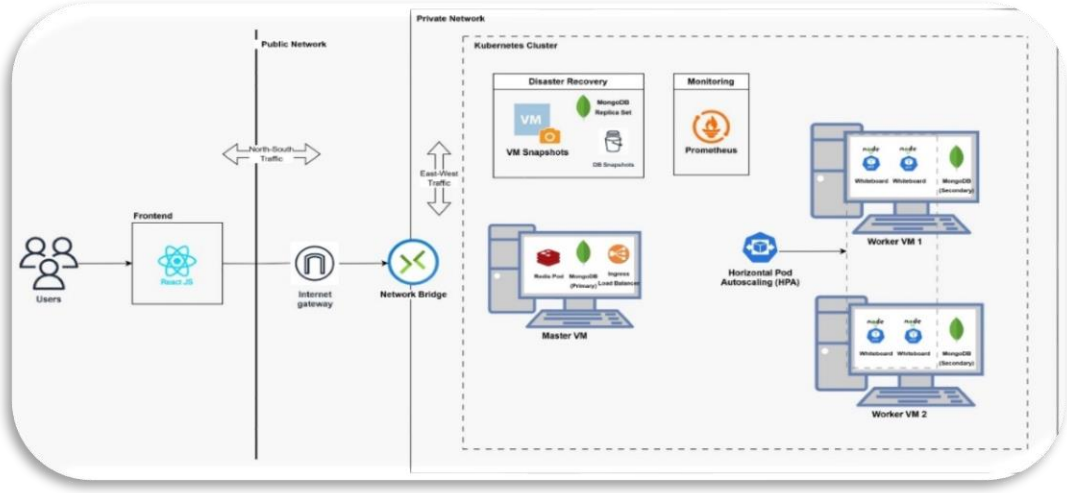


Figure 1(Private Cloud Architecture Design)

Network Topology

Our network topology addresses **Availability** and **Partitioning** from the CAP Theorem [1], ensuring **high availability** and **fault tolerance**. User traffic is routed from the whiteboard frontend through an internet gateway to the private network hosted in our cloud architecture. Oracle VirtualBox manages the Virtual Machines, offering features like snapshots for disaster recovery [2] and a private NAT network, this allows **low latency** as the direct communication within the internal network minimizes the latency associated with external network traffic.

The design includes three VMs running a MicroK8s cluster with one master node and two worker nodes. The master node handles control-plane operations such as cluster orchestration, workload scheduling, and scaling applications. It also hosts the Ingress controller, which load-balances external traffic to worker nodes, ensuring user requests reach the appropriate backend pods.

Horizontal autoscaling creates new pods as demand increases, distributing them evenly across worker nodes for fault tolerance. If a node fails, pods are rescheduled by the master node to maintain availability. **Redis** ensures eventual consistency by locking operations, allowing sequential updates to the whiteboard.

We use **MongoDB** replica sets as our data storage, which is deployed on each VM, this allows us to save the whiteboard states and reflect it across each instance. On the worker nodes we have secondary MongoDB Pods handles read requests which reduces the load on the primary master node. It **minimizes latency** as it would then read from the nearest available local replica. Finally, **Prometheus** monitors the Kubernetes cluster, tracking pod resource usage and performance [3].

Task 2: Implementation of Private Cloud

Hardware configuration

The **hypervisor** used in this implementation is **Oracle VirtualBox**, this is used to create and manage virtual machines, we use this along with **Vagrant** which automates the provisioning and decommissioning of the VMs with minimal intervention.

```
Vagrant.configure("2") do |config|

  # Configure the first VM (vmmaster1)
  config.vm.define "vmmaster1" do |vmaster1|
    master.vm.box = "ubuntu/bionic64"  # Use Ubuntu 24.04
    master.vm.network "private_network", type: "dhcp"
    master.vm.provider "virtualbox" do |vb|
      vb.memory = "13720"  # 13GB RAM
      vb.cpus = 8          # Optional: Set number of CPUs
      vb.disk :disk, size: "30GB"  # 30GB hard disk
    end
  end

  # Configure the second VM (vmworker1)
  config.vm.define "vmworker1" do |vmworker1|
    worker1.vm.box = "ubuntu/bionic64"
    worker1.vm.network "private_network", type: "dhcp"
    worker1.vm.provider "virtualbox" do |vb|
      vb.memory = "12414"  # 12GB RAM
      vb.cpus = 8
      vb.disk :disk, size: "25GB"
    end
  end

  # Configure the third VM (vmworker2)
  config.vm.define "vmworker2" do |vmworker2|
    worker2.vm.box = "ubuntu/bionic64"
    worker2.vm.network "private_network", type: "dhcp"
    worker2.vm.provider "virtualbox" do |vb|
      vb.memory = "12414"  # 12GB RAM
      vb.cpus = 8
      vb.disk :disk, size: "20GB"
    end
  end

end
```

As we can see in figure 2, this is the Vagrant file which describes the VMs that will be provisioned, in this case we have three VMs, vmaster1(master node), vmworker1 and vmworker2(workers node).

Master Node: 13GB RAM, 8 CPUs, 30GB disk space.

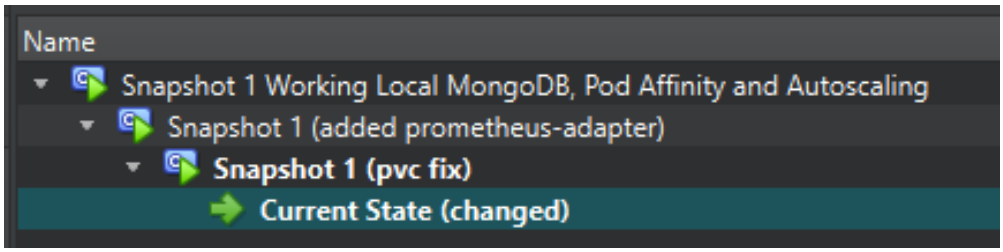
Worker Nodes: 12GB RAM, 8 CPUs, 25GB/20GB disk space.

Operating System: Ubuntu 20.04 for all the VMs.

If we were to add more VMs we can define it in this file to automate the provisioning process.

Figure 2(Vagrant File)

The Oracle VirtualBox Manager allows our architecture to recover from failed implementations or disaster recovery through the use of Snapshots.



Network Configuration

Next, we setup the network configuration, we need to allow the public traffic coming from the whiteboard frontend to communicate with our private network. So ,as we can see in Figure 3, we have the **Bridged Adapter,** this allows the communication between the **public network and private network**. Then we setup the private network as we can see in Figure 4, we setup a **Nat Network** which allows communication between the nodes, this is required for the **Microk8s cluster** to be setup to allow worker nodes to join the master node and work as a cluster.

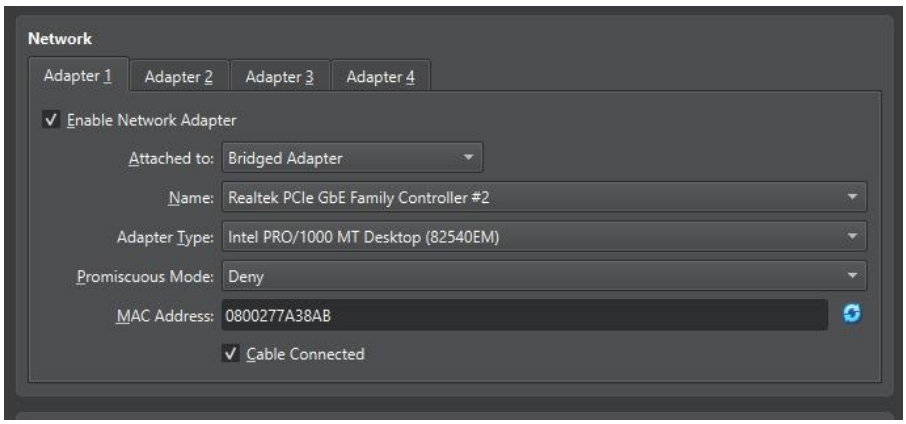


Figure 3 Bridged Adapter

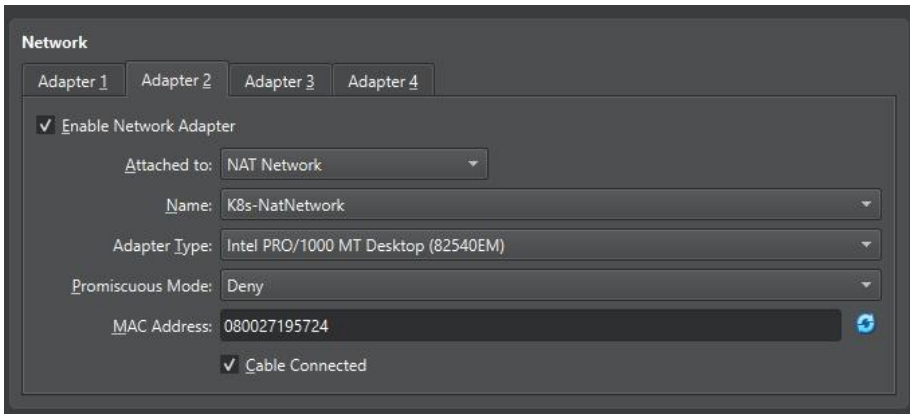


Figure 4 Nat Network

By utilizing this configuration, we **reduce the latency**, as the private network ensures internal communication between the nodes which **minimizes the latency** associated with external network traffic coming from the bridged adapter.

Kubernetes Cluster

The Microk8s cluster requires an **elected node** to be the **master** which is responsible for **orchestrating** all **operations** within the Kubernetes cluster. It **coordinates** the **scheduling** of **workloads, scaling applications, and maintaining** the desired state of the system. So, it acts as the central hub for the Kubernetes cluster.

In our architecture we have three nodes, the **vmaster1** is the **master node,** which schedules tasks to our other nodes **vmworker1** and **vmworker2** which are the worker nodes, this achieves **high availability** as this ensures that if one node fails, the workloads is redistributed to other nodes. We can see those nodes in figure 5.

```
vmaster1@vmmaster1:~$ microk8s kubectl get nodes
NAME         STATUS   ROLES    AGE   VERSION
vmmaster1    Ready    <none>   15d   v1.31.3
vmworker1    Ready    <none>   15d   v1.31.3
vmworker2    Ready    <none>   13d   v1.31.3
```

Figure 5 Microk8s Nodes

To achieve **availability,** we setup a **Horizontal Pod Autoscaler** service which allows for the pods to be rescaled when we met a certain target. We can see this in Figure 6.

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: backend-hpa
  namespace: whiteboard
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: backend
  minReplicas: 4
  maxReplicas: 6
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 9
```

As we can see, we set the target to affect only the **backend deployment** which hosts the **whiteboard instances**, there is a **minimum** number of **replicas** set to 4 as we require two instances to run on the two **vmworkers** by default. The target that triggers the autoscaling is the **CPU average utilization** set to 9**,** this target will be hit when there is load on the CPU, for example a user joining the whiteboard instance and drawing. Once the trigger is set off it will scale the replica pods to 6. This allows for **high availability** as it ensures that if a pod were to fail another one can take over the workload, in other words this is addressing **fault tolerance**.

Figure 6 Horizontal Pod Auto Scaler

```
vmaster1@vmmaster1:~$ microk8s kubectl get pods -n whiteb
NAME                          READY   STATUS    RESTARTS
backend-5dc8749df-9nwsq       1/1     Running   2 (3h7m a
backend-5dc8749df-k7lpl       1/1     Running   3 (3h7m a
backend-5dc8749df-txwp8       1/1     Running   2 (3h7m a
backend-5dc8749df-zw748       1/1     Running   3 (3h7m a
mongodb-0                     1/1     Running   3 (3h8m a
mongodb-1                     1/1     Running   6 (3h9m a
mongodb-2                     1/1     Running   3 (3h9m a
prometheus-5f676fd5c4-mkwk9   1/1     Running   8 (3h9m a
redis-f8db9547-fsbx9          1/1     Running   6 (3h9m a
vmaster1@vmmaster1:~$ microk8s kubectl get hpa -n whitebo
NAME          REFERENCE            TARGETS       MINPODS
backend-hpa   Deployment/backend   cpu: 8%/9%    4
```

Here we can see an example of the running 4 Pods when there is minimal load as indicated in the target percentage.

```
vmaster1@vmmaster1:~$ microk8s kubectl get pods -n whiteboard
NAME                          READY   STATUS    RESTARTS
backend-5dc8749df-2qrz2       1/1     Running   0
backend-5dc8749df-9nwsq       1/1     Running   2 (3h8m ago)
backend-5dc8749df-k7lpl       1/1     Running   3 (3h9m ago)
backend-5dc8749df-txwp8       1/1     Running   2 (3h8m ago)
backend-5dc8749df-vthzq       1/1     Running   0
backend-5dc8749df-zw748       1/1     Running   3 (3h9m ago)
mongodb-0                     1/1     Running   3 (3h10m ago)
mongodb-1                     1/1     Running   6 (3h10m ago)
mongodb-2                     1/1     Running   3 (3h10m ago)
prometheus-5f676fd5c4-mkwk9   1/1     Running   8 (3h10m ago)
redis-f8db9547-fsbx9          1/1     Running   6 (3h10m ago)
vmaster1@vmmaster1:~$ microk8s kubectl get hpa -n whiteboard
NAME          REFERENCE            TARGETS       MINPODS   MAX
backend-hpa   Deployment/backend   cpu: 21%/9%   4         6
```

While here we can see that the pods scale as the load increases as it is indicated by the target percentage.

Figure 7 (Backend Pods)

Partitioning, is achieved through our **Pod Anti-Affinity** and **Node Affinity Rules,** we can see the configuration of this in figure 7**.**

```yaml
spec:
  replicas: 4
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
            - weight: 1
              podAffinityTerm:
                labelSelector:
                  matchLabels:
                    app: backend
                topologyKey: "kubernetes.io/hostname"
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                - key: kubernetes.io/hostname
                  operator: In
                  values:
                  - vmworker1
                  - vmworker2
      containers:
      - name: backend
        image: meharfatimakhan/whiteboard-backend:2.3
        ports:
        - containerPort: 5000
        envFrom:
        - configMapRef:
            name: whiteboard-config
        resources:
          requests:
            cpu: "250m"
            memory: "128Mi"
          limits:
            cpu: "800m"
            memory: "256Mi"
```

The rule sets a minimum of 4 pods by default, so that two instances of the whiteboard application run on two VMs. This is achieved through our **Node Affinity** rule, we have set the scheduling of the pods to be allocated on our two VMs vmworker1 and vmworker2, this ensures a balanced workload across the cluster.

The **Pod Anti-Affinity**, prevents the pods instances from being scheduled on the same node. This guarantees that the application is distributed across different nodes, reducing the risk of a single point of failure further improving **fault tolerance**

Figure 8 Pod Affinity and Node Anti Affinity

To manage the traffic coming from the public network we configured an **Ingress Controller.**

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: backend-ingress
  namespace: whiteboard
spec:
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: backend
            port:
              number: 5000
```

This creates a **single-entry point** used to **route traffic** to our nodes hosting the backend whiteboard instances. This acts as a **Load Balancer** as it distributes traffic across multiple backend pods to ensure balanced resource utilization and avoid overloading any single pod. The Ingress controller is also able to view the state of the pods, so that if unhealthy pod is present, it will not schedule any work which ensures **high availability.**

Figure 6 Ingress Controller

To allow the traffic to be routed correctly we have exposed the IP address and port of the vmmaster1 as this is the entry point for the traffic.

```
vmaster1@vmmaster1: $ microk8s kubectl get nodes -n whiteboard
NAME        STATUS   ROLES    AGE   VERSION   INTERNAL-IP
vmmaster1   Ready    <none>   15d   v1.31.3   192.168.1.134
vmworker1   Ready    <none>   14d   v1.31.3   192.168.1.153
vmworker2   Ready    <none>   12d   v1.31.3   192.168.1.67
```

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: backend
  namespace: whiteboard
  annotations:
    prometheus.io/scrape: "true"
    prometheus.io/port: "5000"
spec:
  selector:
    app: backend
  ports:
  - protocol: TCP
    port: 5000
    nodePort: 30105
    targetPort: 5000
  type: NodePort
```

```
const server = "http://192.168.1.134:30105";
```

Figure 7 Connection route

MongoDB achieves **Availability** and **Partition Tolerance** through the use of **replica sets**. This design allows **MongoDB** to handle node failures and network partitions while ensuring data remains accessible and operations continue.

In our replica set consists one **Primary** and two **Secondary,** the primary residing on our master node and the two secondary on the worker nodes. The primary handles all the write operations and synchronizes the data with the secondaries. The secondaries maintain a copy of the data and serve read operations.

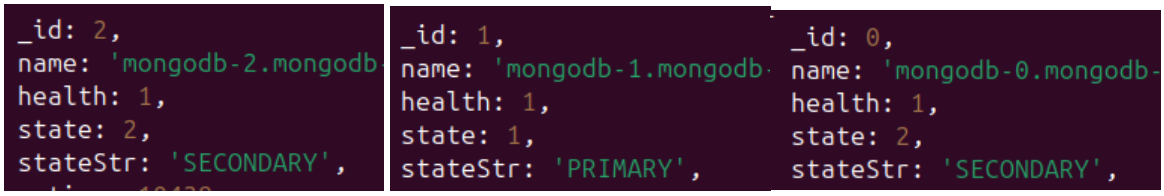


Figure 8 Primary and Secondary Replica Set

This configuration achieves a **Raft-based consensus algorithm** [4]**,** If the primary node fails, the replica set will elect a new primary from the secondary nodes, the system will be briefly unavailable for writes, but availability is restored as a new primary takes over.

The MongoDB pods utilize the Persistent Volumes to store the data, so that if a pod fails and is rescheduled the data persists so that the whiteboard instances stay **consistent**, this proceeds to reattached to the new pod, ensuring High-Availability.

```
vmaster1@vmmaster1:~$ microk8s kubectl get pv -n whiteboard
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM
   AGE
mongodb-pv                                 300Mi      RWO            Retain           Available
   5d12h
pvc-0b8ef82f-304a-44c8-a47e-cce6eb2b2d47   300Mi      RWO            Delete           Bound       whiteboard/mongodb-data-mongodb-2
   5d10h
pvc-a091d985-f4b0-4206-af16-1c4f4898bbf5   300Mi      RWO            Retain           Released    whiteboard/mongodb-pvc
   5d10h
pvc-bb5a1b40-0bf9-4032-bc48-9269897414a0   300Mi      RWO            Delete           Bound       whiteboard/mongodb-data-mongodb-0
   5d10h
pvc-cad99b1f-f484-43d0-a8bb-af18c2dd7393   300Mi      RWO            Delete           Bound       whiteboard/mongodb-data-mongodb-1
   5d10h
pvc-dbb3cf1e-6b14-4c1b-bfd4-6e9fd6cb5cd1   300Mi      RWO            Retain           Bound       whiteboard/mongodb-pvc
```

Figure 9 MongoDB PVs

To monitor our resources usage we implemented **Prometheus**, this allowed monitoring of pods CPU usage and scaling behaviour by utilizing PromQL Queries.

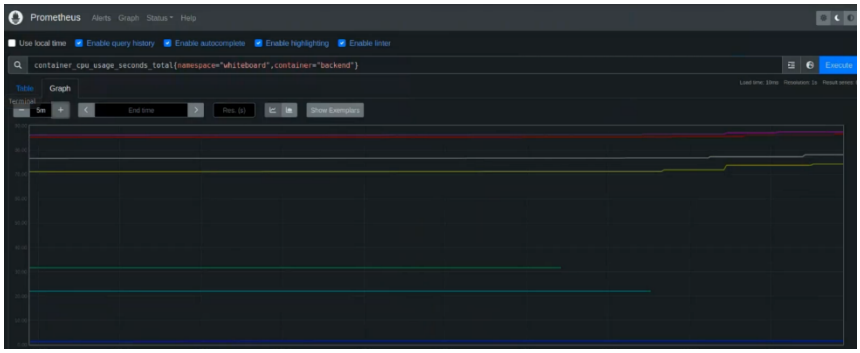


Figure 10 Prometheus

Task 3: Distributed Whiteboard Application

Our Distributed whiteboard application provides an interactive, shared canvas where multiple users can draw and add text simultaneously. It supports basic drawing features such as lines, shapes, and text placement. The whiteboard guarantees that all users see the same state in real time and retrieves the current state for new users joining the session from the MongoDB database.

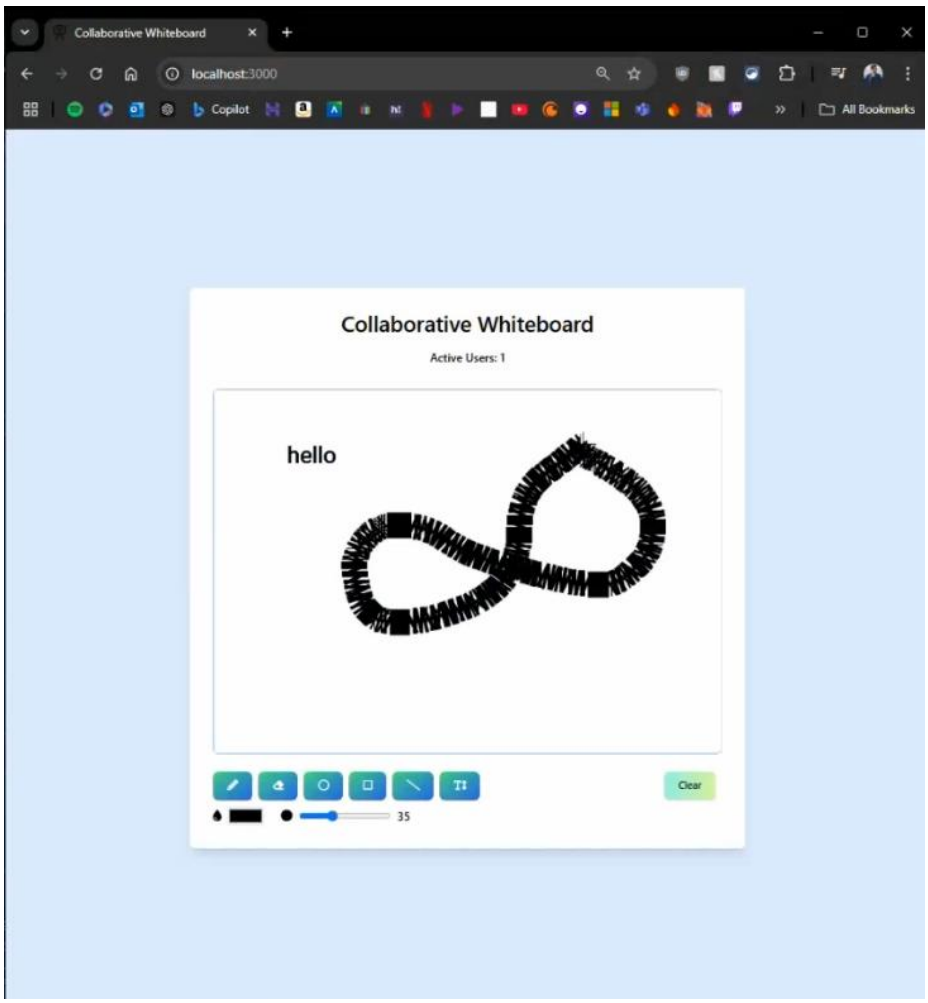


Figure 11 Whiteboard

Implementation of the whiteboard

The application consists of a frontend built with **React.js** and styled using **Tailwind CSS**, offering users a dynamic interface for drawing and adding text. The backend is implemented in **Node.js** with **Express.js**, which manages user interactions, handles session and communication with our **MongoDB** database which handles our storage.

Figure 12 Distributed Whiteboard Design

When a user joins the current state of the whiteboard is obtained as the use of **Persistent Volumes** guarantees that the whiteboard state remains intact and consistent across all users, even during pod rescheduling. **Consistency** is achieved through real-time synchronization through **Socket.IO**, which supports low-latency and updates all users with the same view of the whiteboard. MongoDB's replica sets provide data redundancy and automatic failover, ensuring high availability. **Redis** allows distributed locking which ensures no conflicts when multiple users write at the same time thus achieving **the same state across all users,** as we can see in the figure. *Refer to figure 13 in the appendix*

The whiteboard instances are hosted on two virtual machines within our Kubernetes cluster. *Refer to figure 14 in the appendix*.

Task 4: Private Cloud contribution

My individual contribution consisted of setting up the Oracle VirtualBox VM machines, Vagrant to provision VMs, setting up the **private network** using a NatNetwork to minimize latency and allow communication between VMs. As well as installing the correct dependencies along with Microk8s and deploying the Kubernetes cluster with the integration of **Horizontal Pod AutoScaler** to allow pods to automatically scale, setting the **Pod Anti-Affinity** and **Node Affinity** to equally distribute the workload to our worker VMs, configuring the **MongoDB** replica sets to allow a consensus algorithm to take place by electing a primary node with read and write while secondaries with only read. Lastly, I used **Prometheus** to monitor pod performance, analysing resource utilization to define appropriate thresholds for autoscaling and optimize system efficiency.

Task 5: Critical Review

The system is built on a Kubernetes cluster deployed on three virtual machines using Oracle VirtualBox and Vagrant for virtualization and automation. MicroK8s provides lightweight container orchestration, integrating MongoDB replica sets for data storage, Redis for distributed locking, and Prometheus for monitoring.

MongoDB replica sets ensure **Availability** and **Partitioning** from the CAP Theorem, offering data redundancy and failover through a Raft-based consensus algorithm. Redis enhances consistency by preventing conflicts during simultaneous updates.

Kubernetes features like horizontal pod autoscaling, node affinity, and pod anti-affinity improve fault tolerance and scalability. The Ingress controller manages external traffic, routing requests to backend pods and balancing the load across nodes.

Our system's weaknesses include a heavy reliance on manual configurations, increasing complexity and potential for errors. For example, worker nodes initially failed to join the master node due to naming issues, and MongoDB replica sets required manual initialization, complicating scaling if we were to add more nodes. Redis distributed locking adds slight latency under high concurrency, potentially bottlenecking real-time performance as the number of users grows.

Despite this, the system excels in high availability through Kubernetes and MongoDB failover mechanisms, with Socket.IO enabling real-time collaboration. Horizontal Pod Autoscaler dynamically adjusts backend pods based on CPU usage, ensuring efficient resource utilization. With optimized resources, the system could theoretically handle tens or thousands of users while maintaining availability and partitioning.

To improve our system, we would have to automate the setup or update process of our Kubernetes cluster by using tools like **Terraform** [5], which would streamline the configuration of Kubernetes clusters, MongoDB replica sets, and Redis instances, reducing setup time and minimizing errors.

Optimizing Kubernetes resource requests and container configurations can enhance scalability on limited hardware by preventing over-allocation while maintaining performance. Implementing a write buffer during MongoDB primary failovers could queue write requests, ensuring seamless user experience.

To reduce the reliance on Redis which would be the bottleneck in our system, we could use CRDT (Conflict-Free Replicated Data Types)[6] which could provide locking with low latency and improve real-time performance as more users are introduced.

Reference List

[1] GeeksforGeeks (2024) *The Cap Theorem in DBMS*, *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/the-cap-theorem-in-dbms/ (Accessed: 10 December 2024).

[2] *Oracle VirtualBox*. Available at: https://www.virtualbox.org/manual/ (Accessed: 10 December 2024).

[3] Prometheus *Overview: Prometheus*, *Prometheus Blog*. Available at: https://prometheus.io/docs (Accessed: 11 December 2024).

[4] *GeeksforGeeks (2024) Raft consensus algorithm, GeeksforGeeks. Available at: https://www.geeksforgeeks.org/raft-consensus-algorithm/ (Accessed: 11 December 2024).*

[5]*Terraform registry*. Available at: https://registry.terraform.io/providers/hashicorp/kubernetes/latest/docs (Accessed: 11 December 2024).

[6]*Researchgate | Conflict-Free Replicated Data Types (CRDTs)*. Available at: https://www.researchgate.net/publication/223507377_Intelligent_approaches_using_support_vector_machine_and_extreme_learning_machine_for_transmission_line_protection_Neurocomputing (Accessed: 11 December 2024).

Appendix

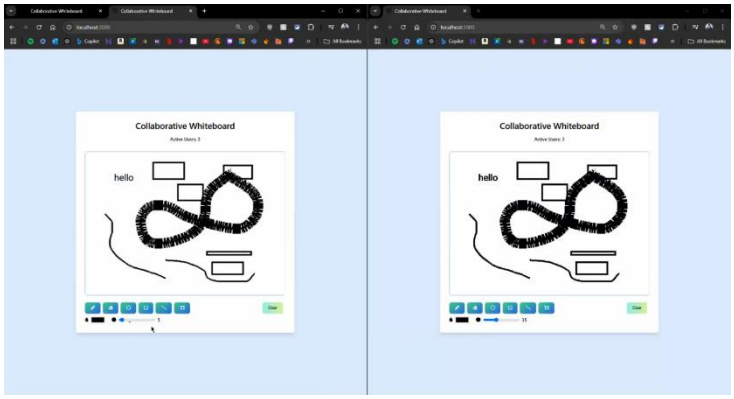


Figure 13 Two instances of Whiteboard

Figure 14 whiteboard backend pods

```
vmaster1@vmmaster1:~$ microk8s kubectl get pods -n whiteboard -o wide
NAME                       READY   STATUS    RESTARTS        AGE     IP            NODE
backend-5dc8749df-k7lpl    1/1     Running   4 (6m25s ago)   16h     10.1.254.51   vmworker1
backend-5dc8749df-mljrp    1/1     Running   0               5m18s   10.1.79.236   vmworker2
backend-5dc8749df-txwp8    1/1     Running   3 (5m44s ago)   16h     10.1.79.248   vmworker2
backend-5dc8749df-zw748    1/1     Running   4 (6m25s ago)   16h     10.1.254.44   vmworker1
```