Saeid Abedi: sabedi4@uwo.ca

Stephan Dhanraj: sdhanraj@uwo.ca

James Zhong: pzhong22@uwo.ca

Reza Motazed Monadjemi: rmotazed@uwo.ca

James Suresh: jsuresh3@uwo.ca

Group 07

Final Project Report: Cloud Computing

Our project is a simple web application that interacts with a deployed database container. This report describes the container developments and deployment to a kubernetes cluster using the GKE platform in both a development and production setting.

Containerized Application

The application consists of two containers. A back-end MongoDB container that allows storage of our relevant data and a front-end Node.js container that handles serving of HTML as well as calls to the database.

Back-end Container

For consistency and ease of use, the application uses the standard released MongoDB image available on dockerhub. This container requires a locally mounted volume to store the data to be accessed. The mounted volume will be covered in our YAML configuration files.

Front-end Container:

The containers serve HTML as well as making the back-end requests to the database. Since Next.js has driver support for MongoDB, the application uses a Next.js server. The front-end container requires building upon the standard node.js image and installing new dependencies including next and MongoClient.

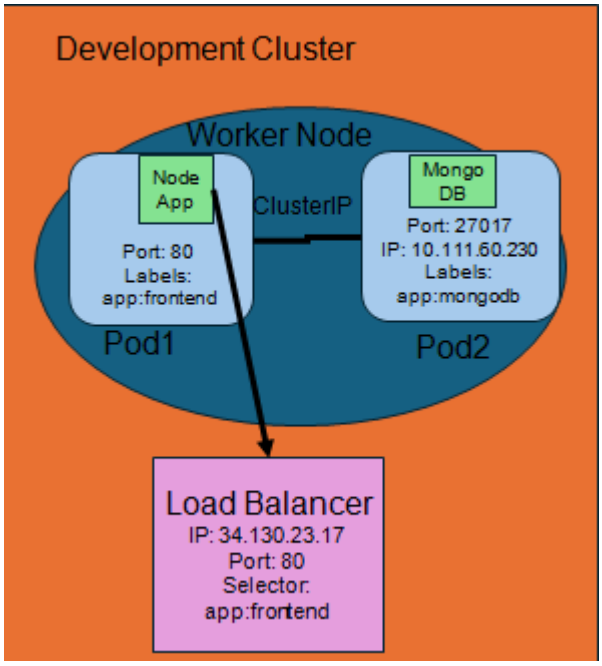
The Front End Dockerfile used is shown in Appendix A. It sets up a Docker container for our Node.js application. It uses Node.js version 21 as the starting point and creates a workspace at /app. After transferring our project's package.json and installing dependencies, it copies the source code into the container. It designates port 80 to listen for web traffic and initiates the application in development mode with npm run dev. This

setup prepares our application to be run in a controlled environment for further development and testing.

The package.json used was shown in appendix B. It outlines the setup for a Node.js application. It names the application, specifies its version, and indicates it's not for public release. It includes commands to run, build, and start the application, using Next.js. Dependencies listed are MongoDB and Mongoose for database operations, and Next.js with React for the web framework. These settings ensure the application can be developed, built, and run with the same configurations every time.

Development Environment

This architecture serves as the testing ground for the application. It is intended to verify container functionality and confirm the application code works.



To deploy in this environment, we create a cluster with only one single node using the default sizing and CPU requirements.

Deployment steps

1. Deploy the cluster

```
gcloud container clusters create dev-cluster-realestate --num-nodes 1 --machine-type e2-standard-4
```

2. Mount the PVC using the PVC yaml file

```
Kubectl apply -f mongodb-pvc.yaml
```

3. Deploy the mongodb database pod

```
Kubectl apply -f db-deployment.yaml
```

4. Create the mongodb clusterIP service

```
Kubectl apply -f mongodb-service_ClusterIP.yaml
```

5. Give the docker image the IP of the cluster via the clusterIP
6. Build the new frontend container with the IP

```
docker build -t northamerica-northeast2-docker.pkg.dev/real-estate-group-416222/docker-rep3/frontend:latest .
```

7. Tag **and** push the new container image

```
docker push northamerica-northeast2-docker.pkg.dev/real-estate-group-416222/docker-rep3/frontend:latest
```

8. Deploy the front-end pod

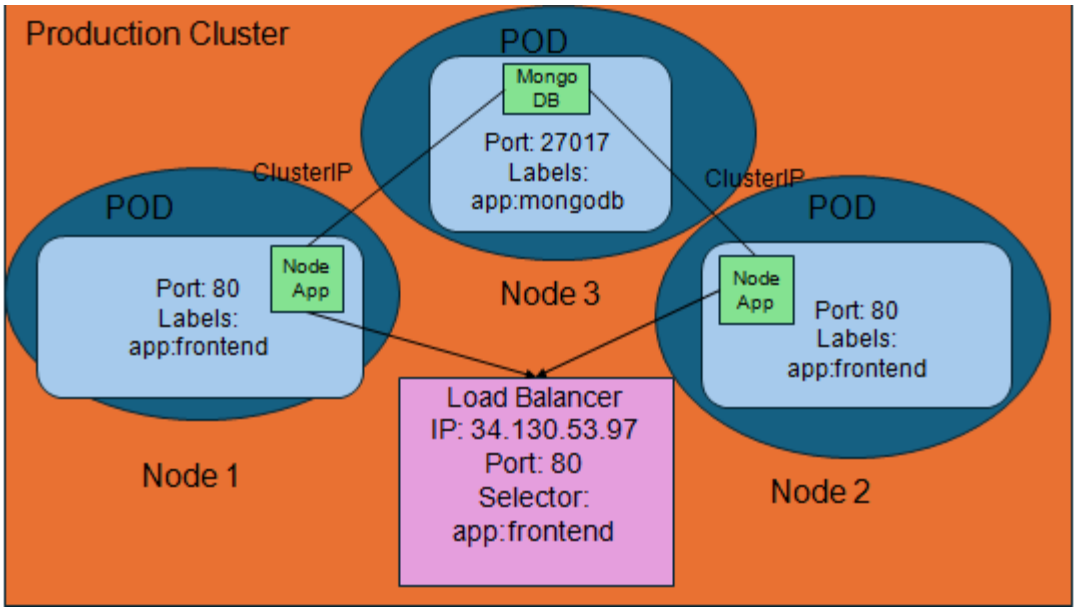
```
kubectl apply -f frontend-deployment.yaml
```

9. Deploy the front end load balancer

```
kubectl apply -f frontend-service.yaml
```

Production Environment

This architecture serves as the deployment environment for the application. It is intended to be the intended cluster for users to access the application.



High Availability:

This architecture is intended for high availability by forcing the pods to deploy onto seperate nodes. This is done through pod anti-affinity using the selectors “frontend” and “mongodb”. This can be seen in Appendix H.

Deployment steps

The Deployment steps for the production environment is almost exactly the same with a different cluster setup and multiple replicas for the frontend pods.

1. Deploy the cluster

```
gcloud container clusters create dev-cluster-realestate --num-nodes 3 --machine-type e2-standard-4
```

2. Mount the PVC using the PVC yaml file

```
Kubectl apply -f mongodb-pvc.yaml
```

3. Deploy the mongodb database pod

```
Kubectl apply -f db-deployment.yaml
```

4. Create the mongodb clusterIP service

```
Kubectl apply -f mongodb-service_ClusterIP.yaml
```

5. Give the docker image the IP of the cluster via the clusterIP
6. Build the new frontend container with the IP

```
docker build -t northamerica-northeast2-docker.pkg.dev/real-estate-group-416222/docker-rep3/frontend:latest .
```

7. Tag **and** push the new container image

```
docker push northamerica-northeast2-docker.pkg.dev/real-estate-group-416222/docker-rep3/frontend:latest
```

8. Deploy the front-end pod

```
kubectl apply -f frontend-deployment.yaml
```

9. Deploy the front end load balancer

```
kubectl apply -f frontend-service.yaml
```

The difference in the deployment is really in the frontend of the production yaml file.

YAML Descriptions

Appendix C shows the Database Deployment Manifest is a Kubernetes yaml file for setting up a MongoDB database. It details a deployment called 'ece9016-mongodb' with one instance that uses the 'mongodb' label. It specifies a container that runs MongoDB on its standard port and includes a username and password for security. It also connects to a persistent storage volume to keep data safe across restarts. This file is the blueprint for deploying a MongoDB service in the cluster.

Appendix D shows the Volume Mounting Manifest which is a request to the Kubernetes system to reserve a persistent storage space for the application. It creates a Persistent Volume Claim (PVC) named 'ece-9016-pvc' that ensures the application can store and retrieve data consistently, even if the container is deleted or moved. The PVC asks for a storage volume with 10 gigabytes of space that can be read from and written to by a single pod at a time, which is what 'ReadWriteOnce' signifies. The 'standard' storage class is specified, which is a default storage class on many Kubernetes clusters, indicating the type of storage to be used. This setup ensures that the necessary data for the application is maintained reliably.

Appendix E shows the Database ClusterIP Service Manifest which is a configuration for setting up a network service in Kubernetes that allows other parts of the application to communicate with the MongoDB database. It creates a service named 'mongodb-service' that doesn't have an external IP address but can be accessed internally within the cluster.

This service listens on port 27017, which is the standard port for MongoDB, and directs any traffic it receives on this port to the MongoDB pod, also on port 27017. It finds the correct pod to send traffic to by looking for a pod with the label 'app: mongodb'. This setup enables other components of the application running in the same Kubernetes cluster to reliably connect and interact with the MongoDB database using a stable internal address.

Appendix F shows the Development Environment Frontend Deployment Manifest which is a configuration file for Kubernetes that describes how to set up the front-end part of the application as a containerized service. It tells Kubernetes to create a deployment named 'frontend' with just one copy running, indicated by 'replicas: 1'. This ensures there's a single instance of the front-end service in the development environment. It uses a label 'app: frontend' to identify the deployment and its pods, making it easier to manage and access them within the Kubernetes system. The deployment is configured to use a specific Docker image for the front-end, located in a Docker repository. This image use the most recent version available. The front-end container is set to listen on port 80, which is the default port for HTTP traffic, allowing web browsers to connect and interact with the application. The 'hostPort' configuration also maps the container's port 80 to the host's port 80, facilitating direct access to the service from the host machine. this file sets up the front-end service in a way that's accessible for testing and development, ensuring that the latest version of the front-end is always deployed in the development environment.

Appendix G shows the Frontend Load Balancer Manifest which is a configuration for setting up a service in Kubernetes that distributes incoming internet traffic across multiple instances of the front-end application. By naming the service 'frontend-service' and specifying it as a 'LoadBalancer', it allows the service to receive external traffic on port 80 (the standard port for web traffic) and forwards it to the front-end application's containers on the same port. This service looks for any pods labeled with 'app: frontend' to direct traffic to, ensuring that requests are evenly spread out, which helps in handling more traffic efficiently and improving the application's availability.

Appendix H shows The Production Environment Frontend Manifest which is a Kubernetes configuration that sets up the front-end portion of your application to run with two instances for higher availability and load distribution. This setup ensures if one instance fails, the other can continue to serve users, improving the application's reliability. It specifies the deployment name as 'frontend' and uses an image from a Docker repository, indicating the application should always use the latest version of this image. Each instance of the front-end listens on port 80, which is typical for web traffic. A key part of this configuration is the 'affinity' setting, which ensures the two instances don't end up on the same physical machine (or node) in the Kubernetes cluster. This is achieved through

'podAntiAffinity', which tells Kubernetes to avoid placing two front-end pods or the mongodb pod on the same host. This spread of pods across different hosts enhances the application's availability and resilience to failures, such as hardware malfunctions, by ensuring that not all instances are affected simultaneously.

Demo Video Link

https://drive.google.com/file/d/1u9sCNFHj-UaH4JHJ1S4JXv_aMBweD59V/view?usp=drive_link

Appendices

Appendix A: Front End Dockerfile

```
FROM node:21

WORKDIR /app

COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 80
CMD npm run dev
```

Appendix B: Frontend package.json

```
{
  "name": "frontend",
  "version": "0.1.0",
  "private": true,
  "scripts": {
    "dev": "next dev -p 80",
    "build": "next build",
    "start": "next start",
    "lint": "next lint"
  },
  "dependencies": {
    "mongodb": "^6.5.0",
    "mongoose": "^8.2.4",
```

```
    "next": "14.1.4",
    "react": "^18",
    "react-dom": "^18"
  }
}
```

Appendix C: Database Deployment Manifest

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ece9016-mongodb
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mongodb
  template:
    metadata:
      labels:
        app: mongodb
    spec:
      nodeSelector:
        kubernetes.io/arch: amd64
      containers:
      - name: mongodb
        image: mongo:latest
        ports:
        - containerPort: 27017
        env:
        - name: MONGO_INITDB_ROOT_USERNAME
          value: adminUser
        - name: MONGO_INITDB_ROOT_PASSWORD
          value: cloudcomputingrocks
        volumeMounts:
        - name: ece-9016-pvc
          mountPath: /data/db # MongoDB default data directory
      volumes:
      - name: ece-9016-pvc
        persistentVolumeClaim:
          claimName: ece-9016-pvc
```

Appendix D: Volume Mounting Manifest

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ece-9016-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: standard
```

Appendix E: Database ClusterIP service Manifest

```
apiVersion: v1
kind: Service
metadata:
  name: mongodb-service
spec:
  type: ClusterIP
  ports:
  - port: 27017
    targetPort: 27017
  selector:
    app: mongodb
```

Appendix F: Development Environment Frontend Deployment Manifest

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
```

```
    spec:
      containers:
        - name: frontend
          image: northamerica-northeast2-docker.pkg.dev/real-
estate-group-416222/docker-rep3/frontend:latest
          ports:
            - containerPort: 80
              hostPort: 80
```

Appendix G: Frontend Load Balancer Manifest

```
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  type: LoadBalancer
  selector:
    app: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

Appendix H: Production Environment Frontend Manifest

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
```

```
        - name: frontend
          image: northamerica-northeast2-docker.pkg.dev/real-
estate-group-416222/docker-rep3/frontend:latest
          ports:
            - containerPort: 80
              hostPort: 80
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                    - mongodb
                    - frontend
              topologyKey: "kubetes.io/hostname
```