1.2 Docker & Kubernetes:

Task: Create a Dockerfile for a simple Node.js application and a Kubernetes deployment manifest yml file to deploy the container.

Deliverable: Provide the Dockerfile, the deployment yml file, and outline the steps to deploy the application to a Kubernetes cluster. In your kubernetes cluster if the pod crashes with OOMKilled error - outline diagnostic steps (how would you troubleshoot and prepare RCA)

Answer:

1. Dockerfile

The following **`Dockerfile`** creates a production-ready Docker image for the React app built with Vite.

```
# Use a lightweight Nginx image
FROM node:18 AS build

WORKDIR /app
COPY package*.json ./
RUN npm install

COPY . .
RUN npm run build

# Use Nginx to serve the app
FROM nginx:alpine

COPY --from=build /app/dist /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

2. Kubernetes Deployment yml

Below is the **`deployment.yml`** for deploying the React app on Kubernetes.

```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: node-app-deployment
  namespace: node-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: node-app
  template:
    metadata:
      name: node-app-dep-pod
      labels:
        app: node-app
    spec:
      containers:
      - name: node-app
        image: docker-username/react-app:v1
        ports:
        - containerPort: 80
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
          limits:
            cpu: 200m
            memory: 256Mi
```

Service Manifest (service.yml)

The following **`service.yml`** creates a **`ClusterIP`** service to expose the React app inside the Kubernetes cluster.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: node-app-service
  namespace: node-app
spec:
  selector:
    app: node-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

NGINX Ingress Manifest (ingress.yml)

The **`ingress.yml`** allows external HTTP traffic to access the React app through an Ingress Controller.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: node-app-ingress
  namespace: node-app
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
    - host: node-app.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: node-app-service
                port:
                  number: 80
```

3. Steps to Deploy the Application to Kubernetes

1. **Build and Push Docker Image**:

Build the Docker image using the **`Dockerfile`**:

```
docker build -t docker-username/react-app:v1 .
```

Log in to Docker Hub:

```
docker login
```

Push the Docker image to your Docker Hub repository:

```
docker push docker-username/react-app:v1
```

2. **Apply Kubernetes Manifests**:

Deploy the application by applying the **`deployment.yml`**, **`service.yml`**, and **`ingress.yml`** files to your Kubernetes cluster:

```
kubectl apply -f deployment.yml
kubectl apply -f service.yml
kubectl apply -f ingress.yml
```

3. **Verify Deployment**:

Check if the pods are running:

```
kubectl get pods
```

Check the status of the services:

```
kubectl get svc
```

Check the ingress status:

```
kubectl get ingress
```

4. Troubleshooting OOMKilled Errors in Kubernetes

If the pod crashes with an **`OOMKilled`** (Out of Memory Killed) error, it means the container has exceeded the memory limit set in the resource configuration. Below are the diagnostic steps to troubleshoot and prepare a Root Cause Analysis (RCA).

Diagnostic Steps:

1. **Check Pod Status**: Use the **`kubectl describe pod <pod-name>`** command to check the pod's status and events, especially to identify the memory issue.

```
kubectl describe pod <pod-name>
```

   Look for the **`OOMKilled`** event in the output. It indicates that the container was terminated because it exceeded its memory limit.

2. **Check Pod Logs**: Check the logs for additional information about why the container was using excessive memory. Use the **`kubectl logs`** command.

```
kubectl logs <pod-name>
```

3. **Review Resource Limits and Requests**: Verify that the memory and CPU limits are appropriately set in the **`deployment.yml`**. If the app is memory-intensive, you may need to increase the memory limit.

```
resources:
  limits:
    memory: "512Mi"  # Consider increasing the memory limit
  requests:
    memory: "256Mi"
```

4. **Check Node Resources**: Ensure that the Kubernetes nodes have sufficient resources to run the application. Use the following command to check node resource utilization:

```
kubectl describe node <node-name>
```

5. **Monitor Memory Usage**: Use Kubernetes monitoring tools (e.g., Prometheus, Grafana, or **`kubectl top`**) to monitor the memory usage of the pods in real-time.

```
kubectl top pod <pod-name>
```

6. **Check for Memory Leaks**: If the application is constantly crashing with OOMKilled, there might be a memory leak in the app. Analyze the application's memory usage and profile the app using debugging tools to identify and fix any leaks.

Root Cause Analysis (RCA)

After gathering the necessary diagnostic data, the RCA should address the following:

- **Exceeding Memory Limits**: Was the memory limit too low for the application's needs? If so, increase the memory limit in the **`resources`** section of the deployment.
- **Memory Leaks in the Application**: If the application has a memory leak, it will eventually consume all allocated memory and be killed. Check for potential issues in the React code or third-party dependencies that could cause excessive memory usage.
- **Insufficient Node Resources**: If the Kubernetes node does not have enough resources to accommodate all pods, it may lead to the pod being terminated. You may need to scale the nodes or distribute the load more efficiently.
- **Configuration Issues**: Ensure the deployment yml specifies appropriate resource requests and limits, and consider optimizing the application to reduce memory consumption.