Devops Engineer Technical Assignment

**Task 1 : Deploy a Systemd Service**

Objective :  Demonstrate the ability to deploy and manage a service on a Linux system.

Step 1 : Create a simple application (e.g., a Python or Node.js HTTP server)

* Create a directory for your application(task.py)

sudo mkdir /opt/mytask

cd /opt/mytask

* Create the Python HTTP server (app.py) with the following content:

# app.py

from http.server import BaseHTTPRequestHandler, HTTPServer

class SimpleHandler(BaseHTTPRequestHandler):

def do\_GET(self):

self.send\_response(200)

self.send\_header('Content-type', 'text/html')

self.end\_headers()

self.wfile.write(b"Hello, World!")

def run(server\_class=HTTPServer, handler\_class=SimpleHandler):

server\_address = ('', 8000)

httpd = server\_class(server\_address, handler\_class)

print("Starting server on port 8000...")

httpd.serve\_forever()

if \_\_name\_\_ == "\_\_main\_\_":

run()

* python3 app.py komutunu çalıştırarak sunucuya erişim yapılabilir.GET isteğini aldığını görmek için ayrı bir ekranda curl <http://20.224.73.223:8000> dediğinizde istek atıyorsa başarılı bir şekilde sunucuya bağlanmıştır.

Step2 : Write a systemd unit file to manage the application as a service.

Systemd manages system services (also called daemons) by starting, stopping, and monitoring them. It handles service dependencies and ensures that services are started in the correct order.Systemd unit file permit to manage the Python app as a service.

* Create a new systemd unit file to define the service

sudo nano /etc/systemd/system/myapp.service

* Add the following contect to the file

[Unit] Description=Simple Python HTTP Server

After=network.target

[Service]

ExecStart=/usr/bin/python3 /opt/myapp/app.py

WorkingDirectory=/opt/myapp

StandardOutput=append:/var/log/myapp.log StandardError=append:/var/log/myapp.log

Restart=always

User=nobody

Group=nogroup

Environment=PYTHONUNBUFFERED=1

[Install]

WantedBy=multi-user.target

Save and close file

Step3 : Enable and Start service

* Reload systemd to read the new unit file

sudo systemctl daemon-reload

* Enable the service so it starts automatically on boot:

sudo systemctl enable myapp.service

Start service

sudo systemctl start myapp.service

Status service

sudo systemctl status myapp.service

metin, ekran görüntüsü, yazı tipi, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu

View logs

cat /var/log/myapp.log

metin, ekran görüntüsü, yazı tipi, çizgi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Task 2 :  Docker-Based Application Deployment**

Objective: Showcase containerization and deployment skills

Step1 : Containerize the application from Task 1 using Docker.

* Create a Dockerfile

To create a Dockerfile , go to path /opt/mytask directory

# Use the official Python image from the Docker Hub

FROM python:3.9-slim

# Set the working directory inside the container

WORKDIR /app

# Copy the application code to the container

COPY app.py /app/

# Expose the port the app will run on

EXPOSE 8000

# Install any required dependencies (if needed)

RUN pip install --no-cache-dir flask

# Define the command to run the app

CMD ["python3", "app.py"]

* Build the docker image

docker build -t mytask:latest .

* Docker run -d -p 8000:8000 mytask:latest

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

We check http GET 200 request server curl <http://20.224.73.223:8000/>

Step2 : Create a docker-compose.yml file to manage the application and a reverse proxy (e.g., NGINX or Traefik) for load balancing

* Create a docker-compose.yml file same directory

/opt/mytask

version: "3.8"

services:

# Python app service

app:

image: mytask:latest

deploy:

replicas: 2 # Set to 2 replicas for high availability

update\_config:

parallelism: 1

restart\_policy:

condition: on-failure

networks:

- app\_network

expose:

- "8000" # Expose port 8000 to the internal network

# NGINX reverse proxy service

nginx:

image: nginx:latest

volumes:

- ./nginx.conf:/etc/nginx/nginx.conf # Load custom NGINX configuration

ports:

- "80:80" # Expose port 80 for external access

networks:

- app\_network

depends\_on:

- app

networks:

app\_network:

driver: bridge

Step3: Ensure high availability by configuring at least 2 replicas.

Create the NGINX Configuration for Load Balancing

We create the NGINX configuration file nginx.conf in the same directory (/opt/mytask)

events {}

http {

upstream app\_servers {

# Load balancing between multiple replicas of the app service

server app:8000;

server app:8000;

}

server {

listen 80;

location / {

proxy\_pass http://app\_servers; # Reverse proxy to the app servers

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header X-Forwarded-Proto $scheme;

}

}

}

Step4 : Run the services with docker compose

docker compose up -d

docker-compose ps

metin, ekran görüntüsü, yazı tipi, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu

We see two replicas (mytask-app-1 , mytask-app-2)

mytask-nginx (reverse Proxy)

We check load balancing to visit <http://20.224.73.223.Refresh> the page a few times.You should see that the response is served from one of the two available

Replicas , ensuring high availability.

**Task 3 :  Kubernetes Cluster Setup**

Objective: Validate experience with Kubernetes and high-availability configurations

Step1 : Deploy the application on a Kubernetes cluster.

apiVersion: apps/v1

kind: Deployment

metadata:

name: mytask

spec:

replicas: 2 # Number of replicas for high availability

selector:

matchLabels:

app: mytask

template:

metadata:

labels:

app: mytask

spec:

containers:

- name: mytask

image: mytask:latest # Replace with your container image name

ports:

- containerPort: 8000

We set replicas: 2 to ensure high availability

Container image : Replace mytask:latest with your actual image name

Ports : The app exposes port 8000 internally

* Create a kubernetes service

apiVersion: v1

kind: Service

metadata:

name: mytask-service

spec:

selector:

app: mytask

ports:

- protocol: TCP

port: 80

targetPort: 8000 # App is running on port 8000

type: ClusterIP

This service exposesport 80 and forwards traffic to the apps internal port 8000.

* Apply the deployment and service

If you are not using minikube , you should also install minikube.For installation minikube , you need at least minimum 2 CPU on machine.

minikube is local Kubernetes, focusing on making it easy to learn and develop for Kubernetes.

For installation step minikube ;

curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube\_latest\_amd64.deb

sudo dpkg -i minikube\_latest\_amd64.deb

To deploy the app to Kubernetes use kubectl to apply both the deployment and service

minikube start

Step2

Then ;

kubectl apply -f app-deployment.yaml



kubectl apply -f app-service.yaml



Step3 Ensure the application is reachable via a LoadBalancer or Ingress

If you're running Kubernetes in an environment that supports external load balancers (e.g., AWS, GCP, Azure), you can use a LoadBalancer service.

apiVersion: v1

kind: Service

metadata:

name: mytask-service

spec:

selector:

app: mytask

ports:

- protocol: TCP

port: 80

targetPort: 8000

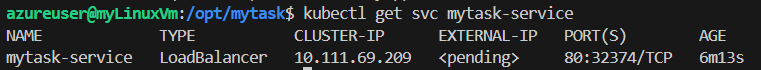
type: LoadBalancer # Expose the service via a LoadBalancer

* Apply to update service

kubectl apply -f app-service-loadbalancer.yaml



kubectl get svc mytask-service



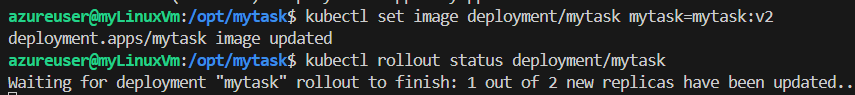
This command update the mytask container in the deployment to use new image (v2)

kubectl set image deployment/mytask mytask=mytask:v2

Step4 Demonstrate rolling updates for the application.

This command show rollout update status

kubectl rollout status deployment/mytask



If something goes wrong with the update you can easily rollback to previous version with this command

kubectl rollout undo deployment/mytask



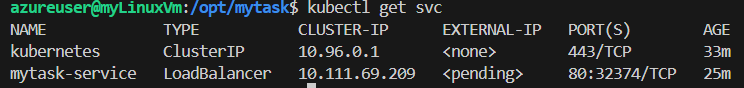
* Verify and Test the App

To check your pods are running correctly

kubectl get pods

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

kubectl get svc 

**Task 4:  Debugging and Troubleshooting**

Objective: Assess problem-solving skills. Scenario: You are provided with a misconfigured systemd service or Kubernetes Deployment

Step1:  Identify and fix the issues.

Scenario1 : Asume that you're working with a **systemd** service for a Python application, and the service isn't starting correctly.For example , our service is mytask.service

[Unit] Description=Python Application

After=network.target

[Service] ExecStart=/usr/bin/python3 /opt/mytask/app.py

WorkingDirectory=/opt/mytask

StandardOutput=append:/var/log/mytask.log StandardError=append:/var/log/mytask.log

Restart=always

[Install]

WantedBy=multi-user.target

There are many issue init file ;

1. No user/group configuraion : The service file doesn’t specify which user or group the service should run as. Running services as the root user is risky and might cause permission issues when accessing files
2. Incorrect Log File Permissions
3. Missing ExecStart Path

We update mytask.service file

[Unit]

Description=Simple Python HTTP Server

After=network.target

[Service]

ExecStart=/usr/bin/python3 /opt/mytask/app.py

WorkingDirectory=/opt/mytask

StandardOutput=append:/var/log/myapp.log

StandardError=append:/var/log/myapp.log

Restart=always

User=nobody

Group=nogroup

Environment=PYTHONUNBUFFERED=1

[Install]

WantedBy=multi-user.target

Then apply fix steps

Reload systemd to apply changes

sudo systemctl daemon-reload

Correct permission log file exists

sudo touch /var/log/mytask.log

sudo chown nobody:nogroup /var/log/mytask.log

sudo chmod 664 /var/log/mytask.log

Ensure the python scripty executable

sudo chmod +x /opt/mytask/app.py

Restart the service

sudo systemctl restart mytask.service

sudo systemctl status mytask.service

check logs

cat /var/log/mytask.log

**Task 4:  Talos-Focused Configuration (Optional, if Talos is critical)**

Objective: Demonstrate familiarity with Talos

Talos is a modern, secure, and minimal operating system designed specifically for running Kubernetes clusters. It's built for automation, offering a streamlined approach to managing Kubernetes nodes without the overhead of a traditional operating system.

Step1 : Explain how you would manage and troubleshoot a Talos-based cluster

First you should install talosctl;

curl -sL https://talos.dev/install | sh

Create talosconfig.yml to same path in this directory

cluster:

name: my-cluster

controlPlaneEndpoint: "https://192.168.49.2:6443"

network:

interfaces:

- interface: eth0

dhcp: true

type: "ethernet"

routes:

- to: 0.0.0.0/0

via: 10.1.1.4

nodes:

- name: minikube

role: control-plane

ip: 192.168.49.2

os: linux

cpu: 2

memory: 4096

network: eth0

containerRuntime: containerd

k8sVersion: v1.22.0

storage:

root:

size: 20Gi

* Create talosctl cluster

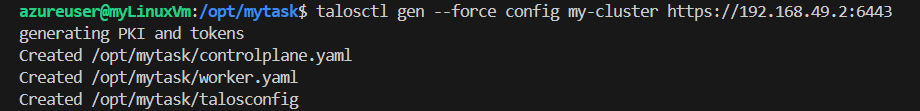
talosctl cluster create

talosctl cluster show

metin, ekran görüntüsü, yazı tipi, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu

talosctl gen --force config my-cluster <https://192.168.49.2:6443>



This generates a config for the cluster that includes control plane and worker nodes.