Kubernetes Pod Autoscaling with Flask and HPA

Objective

To implement a scalable Kubernetes deployment using Horizontal Pod Autoscaler (HPA) that adjusts the number of pods based on CPU usage of a Flask-based web application.

Folder Structure

```
flask-hpa-k8s/
├── app.py                # Flask app with a CPU load endpoint
├── requirements.txt      # Flask dependencies
├── Dockerfile            # For building container image
├── load_generator.py     # Script to simulate heavy CPU load
├── k8s/
│   ├── deployment.yaml   # Kubernetes Deployment for Flask app
│   ├── service.yaml      # Kubernetes Service (NodePort)
│   └── hpa.yaml          # Horizontal Pod Autoscaler config
└── README.md             # Project guide and documentation
```

Technologies Used

- Python 3, Flask
- Docker
- Kubernetes (Minikube)
- Horizontal Pod Autoscaler (HPA)
- Metrics Server (enabled in Minikube)

Step-by-Step Implementation

1. Build Docker Image

docker build -t flask-hpa-app .

2. Load Docker Image into Minikube

minikube image load flask-hpa-app

3. Apply Kubernetes Deployment and Service

kubectl apply -f k8s/deployment.yaml
kubectl apply -f k8s/service.yaml

4. Get the Service URL

minikube service flask-service --url

5. Apply Horizontal Pod Autoscaler

kubectl apply -f k8s/hpa.yaml

6. Trigger Load (Simulate Traffic)

python load_generator.py

7. Monitor HPA Activity and Pod Scaling

kubectl get hpa -w
kubectl get pods -w

Expected Behavior

- The application initially runs with one pod.
- When CPU usage exceeds 50%, the HPA increases the number of pods.
- After the load ends and CPU usage drops, the number of pods automatically scales back down.

Accessing the Application

Use the service URL returned by:
minikube service flask-service --url
Example:

- *http://<minikube-ip>:<node-port>/ returns "Hello from Flask!"*
- *http://<minikube-ip>:<node-port>/load triggers CPU load*

Architectural Decisions

1. **Flask Web App**
   Flask was chosen for its simplicity and ease of demonstration in CPU-bound workloads.
2. **Docker Image**
   A lightweight image (python:3.9-slim) was used for quick build and deployment.
3. **Kubernetes Resource Limits**
   CPU request was set to 100m and limit to 500m to help trigger HPA scaling behavior.
4. **NodePort Service**
   NodePort allowed easy access to the Flask app locally without using an ingress controller.
5. **Horizontal Pod Autoscaler**
   Configured to scale from 1 to 10 pods based on 50% average CPU utilization.

Challenges Faced

Area	Challenge	Solution
Metrics	kubectl top pods showed no output initially	Installed and enabled metrics-server in Minikube
HPA Delay	Scaling response took time	Observed delay of 30–60 seconds before metrics registered
CPU Load	Load script didn't stress CPU enough	Rewrote load logic using nested loops and delay logic
Port Access	App not accessible from host	Verified correct NodePort and used minikube service
Scaling Down	Pods remained high after load ended	Observed cooldown period of 5–6 minutes before downscale

Scope for Improvement

1. **Memory-Based Autoscaling**
   Extend the HPA configuration to monitor memory usage in addition to CPU.
2. **Monitoring Tools**
   Integrate Prometheus, Grafana, or Kubernetes Dashboard for visual insights and alerts.
3. **Ingress Controller**
   Replace NodePort with NGINX Ingress for clean and scalable URL routing.
4. **Custom Metrics Autoscaling**
   Use Prometheus Adapter to scale based on request rate, latency, or other custom business logic.
5. **Production Readiness**
   - Add liveness and readiness probes
   - Enable rolling deployments
   - Deploy on managed cloud Kubernetes platforms (like AWS EKS, GCP GKE)

Conclusion

This project successfully demonstrates Kubernetes-based autoscaling using Horizontal Pod Autoscaler. By simulating CPU load and watching pods scale dynamically, it provides a solid foundation for building resilient and scalable microservices in real-world environments.