Lab 9 - Autoscaling Models

Step 1: Install KEDA

```
helm repo add kedacore https://kedacore.github.io/charts
helm repo update

helm install keda kedacore/keda \
  --namespace keda \
  --create-namespace
```

validate

```
kubectl get all -n keda
```

Step 2: Add Resource Spec to the Pod

`File: deployment/kubernetes/model-deploy.yaml`

```
    spec:
      containers:
      - image: xxxxxx/house-price-model:latest
        name: house-price-model
        ports:
        - containerPort: 8000
        # Add the following spec
        resources:
          requests:
            cpu: "50m"
            memory: "64Mi"
          limits:
            cpu: "100m"
```

```
            memory: "128Mi"
```

```
kubectl apply -k deployment/kubernetes
```

Step 3: Pick Custom Metric (e.g. Latency 95th)

From your dashboard JSON file used earlier to create the Grafana Dashboard, you could pick the custom metric queries, which could could also test with prometheus server at http://localhost:30300/query

e.g.

```
histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket[1m]))
by (le, handler))
```

You can simplify for total latency across all handlers:

```
histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket[1m]))
by (le))
```

Let's say you want to scale up if **p95 latency > 0.5 seconds**.

Step 4: Create a KEDA ScaledObject

File: `fastapi-scaledobject.yaml`

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: fastapi-latency-autoscaler
```

```
  namespace: default
spec:
  scaleTargetRef:
    name: model  # update to your actual deployment name
  minReplicaCount: 1
  maxReplicaCount: 5
  pollingInterval: 30  # seconds
  cooldownPeriod: 300  # seconds before scaling down
  triggers:
    - type: prometheus
      metadata:
        serverAddress: http://prom-kube-prometheus-stack-
prometheus.monitoring.svc:9090
        metricName: fastapi_latency_p95
        query: |
          histogram_quantile(0.95,
sum(rate(http_request_duration_seconds_bucket[1m])) by (le))
        threshold: "0.5"
```

Step 5: Apply the ScaledObject

```
kubectl apply -f fastapi-scaledobject.yaml
```

```
kubectl get scaledobject
kubectl get hpa  # KEDA will create a linked HPA here
```

Step 6: Add Another Metric (Multi-Trigger Scaling)

Add autoscaling based on request_rate (number of simultaneous inference requests) as well

```
triggers:
  - type: prometheus
    metadata:
```

```
      serverAddress: http://...:9090
      metricName: latency
      query: ...histogram_quantile...
      threshold: "0.5"
  # Add only the following to existing code
  - type: prometheus
    metadata:
      serverAddress: http://prom-kube-prometheus-stack-
prometheus.monitoring.svc:9090
      metricName: request_rate
      query: sum(rate(http_requests_total[1m]))
      threshold: "1000"
```

apply

```
kubectl apply -f fastapi-scaledobject.yaml
```

validate

```
kubectl get scaledobject,hpa,pods
```

Validate and Monitor

- Watch KEDA logs: `kubectl logs -n keda deploy/keda-operator`
- See autoscaling in action via `kubectl get pods`

Step 7: Run a Load Test

Install hey – Minimal HTTP load generator

✅ **Installation**

- **macOS**:

`brew install hey`

- **Linux**:

`sudo snap install hey`

or

`go install github.com/rakyll/hey@latest`

- **Windows**:
  Download from GitHub releases and add to PATH.

Create a json file to send data for predictions

File : `predict.json`

```
{
  "sqft": 4500,
  "bedrooms": 4,
  "bathrooms": 2,
  "year_built": 2014,
  "condition": "Good",
  "location": "Urban"
}
```

Try one prediction

```
curl -X POST http://localhost:30100/predict \
  -H "Content-Type: application/json" \
  -d @predict.json
```

Run a longer load test

```
hey -n 5000 -c 200 -m POST \
  -H "Content-Type: application/json" \
```

```
  -D predict.json \
  http://localhost:30100/predict
```

You could also run a load test with an interval e.g. (-z 3m)

```
hey -z 3m -c 200 -m POST \
  -H "Content-Type: application/json" \
  -D predict.json \
  http://localhost:30100/predict
```

[sample output]

```
Summary:
  Total:5.9189 secs
  Slowest:  0.5623 secs
  Fastest:  0.0050 secs
  Average:  0.2070 secs
  Requests/sec: 844.7576

  Total data:   725000 bytes
  Size/request: 145 bytes

Response time histogram:
  0.005 [1] |
  0.061 [157]   |■■■■
  0.116 [230]   |■■■■■■
  0.172 [1628]  |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.228 [1032]  |■■■■■■■■■■■■■■■■■■■■■■■■■
  0.284 [1203]  |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.339 [444]   |■■■■■■■■■■■
  0.395 [147]   |■■■■
  0.451 [69]|■■
  0.507 [60]|■
  0.562 [29]|■


Latency distribution:
  10% in 0.1250 secs
  25% in 0.1436 secs
```

```
  50% in 0.1951 secs
  75% in 0.2589 secs
  90% in 0.3030 secs
  95% in 0.3541 secs
  99% in 0.4757 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0005 secs, 0.0050 secs, 0.5623 secs
  DNS-lookup:   0.0001 secs, 0.0000 secs, 0.0114 secs
  req write:0.0000 secs, 0.0000 secs, 0.0058 secs
  resp wait:0.2062 secs, 0.0044 secs, 0.5622 secs
  resp read:0.0002 secs, 0.0000 secs, 0.0231 secs

Status code distribution:
  [200] 5000 responses
```

Troubleshooting : Configuring Seprate Endpoint for FastAPI Metrics

Add the following code to the FastAPI App somewhere between `app = FastAPI(` and the endpoint configurations.

File `src/api/main.py`

```
from prometheus_client import start_http_server
import threading


# Start Prometheus metrics server on port 9100 in a background thread
def start_metrics_server():
    start_http_server(9100)
```

Also update `Doclerfile` in the root of the project dir to expose `9100` port

File `Dockerfile`

```
EXPOSE 8000 9100
```

Once updated, commit these files to have the GitHub Actions Pipeline rebuild the container image and publish it to the registry.

Also update the kubernetes manifest with metrics endpoint

File : `deployment/kubernetes/model-svc.yaml`

```
spec:
  ports:
  - name: "8000"
    nodePort: 30100
    port: 8000
    protocol: TCP
    targetPort: 8000
  - name: metrics
    port: 9100
    targetPort: 9100
```

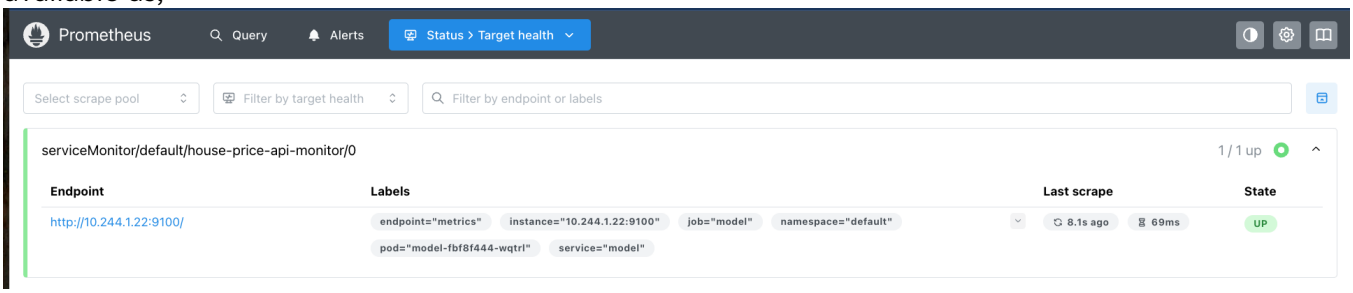
and service monitor to scrape this new endpoint

File: `deployment/monitoring/servicemonior.yaml`

```
spec:
  selector:
    matchLabels:
      app: model
  namespaceSelector:
    matchNames:
      - default  # or your namespace
  endpoints:
    - port: metrics
      path: /
      interval: 15s
      scrapeTimeout: 10s
```

and apply these manifests.

After a few minute, if you check prometheus with `/targets` path you should see the endpoint being available as,



PART II

Adding CPU Based Scaling

Setup Metrics Server using

```
cd ~
git clone https://github.com/schoolofdevops/metrics-server.git
kubectl apply -k metrics-server/manifests/overlays/release
```

Give it a couple of minutes to setup and collect metrics. Then validate by running,

```
kubectl top pods
kubetl top nodes
```

Update the `fastapi-scaledobject.yaml` to scale on CPU Metric as ,

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
```

```
metadata:
  name: fastapi-latency-autoscaler
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: model  # update to your actual deployment name
  minReplicaCount: 1
  maxReplicaCount: 5
  pollingInterval: 30  # seconds
  cooldownPeriod: 300  # seconds before scaling down
  triggers:
    - type: prometheus
      metadata:
        serverAddress: http://prom-kube-prometheus-stack-
prometheus.monitoring.svc:9090
        metricName: fastapi_latency_p95
        query: |
          histogram_quantile(0.95,
sum(rate(http_request_duration_seconds_bucket[1m])) by (le))
        threshold: "0.5"
    - type: cpu
      metricType: Utilization # Allowed types are 'Utilization' or
'AverageValue'
      metadata:
        value: "50"
```

source: KEDA Autoscaler with Latency and CPU Utilization

```
kubectl apply -f fastapi-scaledobject.yaml
kubectl get scaledobject,hpa,pods
```

Now run the load test again and check if it scales.

Using VerticlePodAutoscaler

Before you begin, ensure OpenSSL is installed and is up to date on your system.

Clone the kubernetes/autoscaler GitHub repository and switch to path where VPA manifests are,

```
git clone https://github.com/kubernetes/autoscaler.git
cd autoscaler/vertical-pod-autoscaler/
```

deploy VPA as

```
./hack/vpa-up.sh
```

Create a VPA Policy

File: `model-vpa.yaml`

```
---
apiVersion: "autoscaling.k8s.io/v1"
kind: VerticalPodAutoscaler
metadata:
  name: model
  labels:
    role: model
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: model
  updatePolicy:
    updateMode: "Auto"
  resourcePolicy:
    containerPolicies:
```

```
      - containerName: '*'
        minAllowed:
          cpu: 50m
          memory: 64Mi
        maxAllowed:
          cpu: 500m
          memory: 512Mi
        controlledResources: ["cpu", "memory"]
```

Source: model-vpa.yaml

apply

```
kubectl apply -f model-vpa.yaml
```

watch

```
kubectl get vpa model --watch
```

[sample output]

```
NAME    MODE    CPU    MEM    PROVIDED    AGE
vote    Auto                              5s
vote    Auto    50m    262144k    True        30s
```

Run the load test again to check if its dynamically scaling resources.

Reference :

- ScaledObject Spec : ScaledObject specification

#courses/mlops