What we’re learning

1. HPA - Horizontal Pod Autoscaling
2. Node Autoscaling
3. Resource management

Horizontal pod accelerator

Ref - https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

A Horizontal Pod Autoscaler (HPA) is a Kubernetes feature that automatically adjusts the number of pod replicas in a deployment, replica set, or stateful set based on observed metrics like CPU utilisation or custom metrics.

This helps ensure that the application can handle varying loads by scaling out (adding more pod replicas) when demand increases and scaling in (reducing the number of pod replicas) when demand decreases.

Horizontal scaling

As the name suggests, if you add more pods to your cluster, it means scaling `horizontally`. Horizontally refers to the fact that you havent increased the `resources` on the machine.

Architecture

Kubernetes implements horizontal pod autoscaling as a `control loop` that runs intermittently (it is not a continuous process) (once every 15s)

- cadvisor - https://github.com/google/cadvisor
- Metrics server - The Metrics Server is a lightweight, in-memory store for metrics. It collects resource usage metrics (such as CPU and memory) from the kubelets and exposes them via the Kubernetes API (Ref - https://github.com/kubernetes-sigs/metrics-server/issues/237)

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/d
or
Apply from here - https://github.com/100xdevs-cohort-2/week-28-manifests
```

- Try getting the metrics

```
kubectl top pod -n kube-system
kubectl top nodes -n kube-system
```

Sample request that goes from hpa controller to the API server

```
GET https://338eb37e-2824-4089-8eee-5a05f84fb85e.vultr-k8s.com:6443/apis/metrics.k8s
```

App for the day

We'll be creating a simple express app that does a CPU intensive task to see horizontal scaling in action.

```
import express from 'express';

const app = express();
const BIG_VALUE = 10000000000;

app.get('/', (req, res) => {
    let ctr = 0;
    for (let i = 0; i < BIG_VALUE; i++) {
        ctr += 1;
    }
    res.send('Hello World!');
});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

The app is deployed at https://hub.docker.com/r/100xdevs/week-28

Creating the manifests

Hardcoded replicas

Lets try to create a deployment with `hardcoded` set of replicas

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cpu-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: cpu-app
  template:
    metadata:
      labels:
        app: cpu-app
    spec:
      containers:
      - name: cpu-app
        image: 100xdevs/week-28:latest
        ports:
        - containerPort: 3000
```

- Create a serice

```yaml
apiVersion: v1
kind: Service
metadata:
  name: cpu-service
spec:
  selector:
    app: cpu-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
  type: LoadBalancer
```

With a horizontal pod accelerator

- Add HPA manifest

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: cpu-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: cpu-deployment
  minReplicas: 2
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

- Apply all three manifests

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

💡 You can scale up/down based on multiple metrics.
If either of the metrics goes above the threshold, we scale up
If all the metrics go below the threshold, we scale down

Scaling up

Before we load test, add some resource limits to your pods. We're doing this to get around this error - https://github.com/kubernetes-sigs/metrics-server/issues/237

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cpu-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: cpu-app
  template:
    metadata:
      labels:
        app: cpu-app
    spec:
      containers:
      - name: cpu-app
        image: 100xdevs/week-28:latest
        ports:
        - containerPort: 3000
        resources:
          requests:
            cpu: "100m"
          limits:
            cpu: "1000m"
```

Copy

- Try sending a bunch of requests to the server (or just visit it in the browser)

```
npm i -g loadtest
loadtest -c 10 --rps 200 http://65.20.89.70
```

Copy

- Check the CPU usage

```
kubectl top pods
```

Copy

- See the hpa average usage

```
 kubectl get hpa
```

Copy

- Check the number of pods, see as they scale up

```
kubectl get pods
```

Copy

Formula for scaling up

Resource requests and limits

Ref - https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

When you specify a Pod, you can optionally specify how much of each resource a container needs. The most common resources to specify are CPU and memory (RAM).

There are two types of resource types

Resource requests

The kubelet reserves at least the `request` amount of that system resource specifically for that container to use.

Resource limits

When you specify a resource *limit* for a container, the kubelet enforces those limits so that the running container is not allowed to use more of that resource than the limit you set.

Difference b/w `limits` and `requests`

If the node where a Pod is running has enough of a resource available, it's possible (and allowed) for a container to use more resource than its `request` for that resource specifies. However, a container is not allowed to use more than its resource `limit`.

Experiments

▼ 30% CPU usage on a single threaded Node.js app

Update the spec from the last slide to decrease the CPU usage. Notice that the CPU doesnt go over 30% even though this is a Node.js app where it can go up to 100%

Copy

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cpu-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: cpu-app
  template:
    metadata:
      labels:
        app: cpu-app
    spec:
      containers:
      - name: cpu-app
        image: 100xdevs/week-28:latest
        ports:
        - containerPort: 3000
```

```
          resources:
            requests:
              cpu: "100m"
            limits:
              cpu: "300m"
```

Try hitting the server

▼ Request 2 vCPU in 10 replicas

Try requesting more resources than available in the cluster.

Copy

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cpu-deployment
spec:
  replicas: 10
  selector:
    matchLabels:
      app: cpu-app
  template:
    metadata:
      labels:
        app: cpu-app
    spec:
      containers:
      - name: cpu-app
        image: 100xdevs/week-28:latest
        ports:
        - containerPort: 3000
        resources:
          requests:
            cpu: "1000m"
          limits:
            cpu: "1000m"
```

Cluster autoscaling

Ref - https://github.com/kubernetes/autoscaler

Cluster Autoscaler - a component that automatically adjusts the size of a Kubernetes Cluster so that all pods have a place to run and there are no unneeded nodes. Supports several public cloud providers. Version 1.0 (GA) was released with kubernetes 1.8.

Underprovisioned resources

In the last slide, we saw that we didn't have enough resources to schedule a pod on.

Let's make our node pool dynamic and add a `min` and `max` nodes.

Restart the deployment

```
kubectl delete deployment cpu-deployment
kubectl apply -f deployment.yml
```

Notice a new node gets deployed

Logs of the cluster autoscaler

kubectl get pods -n kube-system | grep cluster-autoscaler

Try downscaling

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cpu-deployment
spec:
  replicas: 10
  selector:
    matchLabels:
      app: cpu-app
  template:
    metadata:
      labels:
        app: cpu-app
    spec:
      containers:
      - name: cpu-app
        image: 100xdevs/week-28:latest
        ports:
        - containerPort: 3000
        resources:
          limits:
            cpu: "1000m"
```

```
            requests:
              cpu: "1000m"
```

Notice the number of server goes down to 2 again

Good things to learn after this -

1. Gitops (ArgoCD)
2. Custom metrics based scaling, event based autoscaling - https://www.giffgaff.io/tech/event-driven-autoscaling
3. Deploying prometheus in a k8s cluster, scaling based on custom metrics from prometheus

Kubernetes Lab

Base repository - https://github.com/code100x/algorithmic-arena/

Things to do -

1. Create a PV, PVC for the postgres database.
2. Create a PV, PVC for redis.
3. Create deployments for redis, postgres.
4. Create ClusterIP services for redis, postgres
5. Create a deployment for the nextjs app, expose it via a loadbalancer service on ‣
6. Create a deployment for the judge api server. Expose it via a ClusterIP service
7. Create a deployment for the judge workers. Add resource limits and requests to it
8. Create a HPA that scales based on the `pending submission queue` length in the redis queue
    1. You can either expose an endpoint that you use as a custom metric
    2. You can put all metrics in prometheus and pick them up from there
    3. You can use KEDA to scale based on redis queue length