Pod Autoscaling with HPA and KEDA

Document Level Classification

300

- Introduction
    - Do I need Pod Autoscaling?
    - What is HPA?
    - What is KEDA?
    - What to use?
- Prerequisites
- Intro to the Horizontal Pod Autoscaler
    - HPA with epic-app
    - HPA with Kubernetes Manifests
- Intro to Kubernetes Event Driven Autoscaling (KEDA)
    - Additional Context
    - Using KEDA with epic-app
        - Controlling HPA scaling behavior via KEDA
    - Using KEDA with Kubernetes Manifests
    - Advanced Examples With KEDA Scalers
        - AWS SQS Queue
            - Creating a AWS SQS Queue
            - Update applications helm values file

    - Working with KEDA Scalers That Require Authentication
- Manually Scaling Pod Deployments
- Next steps
- Tips
- Additional resources

Introduction

Adjusting deployment size (the number of pods) to meet demand is a common operational task. For some workloads, it can make sense to do this automatically based on one or more metrics. For example, if your application is CPU-constrained, you may want to add pods automatically after average CPU across all pods reaches a certain threshold. At Epic there are two options available for configuring pod autoscaling: HPA and KEDA. This doc is designed to give you an introduction to both using some practical examples.

Do I need Pod Autoscaling?

This is an application-specific question. Autoscaling can be a useful tool if you need to regularly grow and reduce the number of pods based on well-understood criteria, such as application load or other metrics. It is not a magic bullet, and simply turning it on without an analysis of your application behavior and requirements can have negative effects. Some things to think about:

- Are you regularly scaling up and down based on some known metric (or do you anticipate this need)?
- Is your application capable of gracefully handling pods starting up and shutting down at arbitrary times?

If the answer to these questions is no, you might do more harm than good with autoscaling enabled. Service owners are advised to consult within their team to determine if enabling autoscaling is worthwhile.

What is HPA?

HPA stands for Horizontal Pod Autoscaler, and it is a built-in solution in Kubernetes. HPA can be configured to automatically scale an application (*Deployment or StatefulSet*) based on pod CPU utilization or pod memory utilization. It can also be used to scale based on custom metrics.

What is KEDA?

KEDA is the Kubernetes-based Event Driven Autoscaler, and it is something we have added to all EKS clusters we support. KEDA works together with the HPA to provide additional scaling functionality, such as scaling based on events, scaling pods down to zero, and scaling on two or more metrics. KEDA supports more than 50 types of scalers - you can read more about them here.

What to use?

This depends on your use case, but in general, we recommend you use HPA if you need to scale based on CPU or Memory utilization. For more complex scaling requirements, KEDA provides the most options. While we can't explore them all in this doc, we will take a look at one example below where scaling is done based on Amazon SQS messages.

Prerequisites

This doc assumes some familiarity with Kubernetes deployments using helm, including authenticating and deploying to a specific cluster**.** See Introduction to Helm Deployments if you need an introduction to basic deployment concepts. You'll also need the following tools installed locally for this doc:

- Helm 3.x
- kubectl

Intro to the Horizontal Pod Autoscaler

HPA with epic-app

Configuring the HPA controller for your deployment with requires just a few inputs. You'll need to supply a **threshold,** that is, the average CPU or memory utilization value that will be used to determine when to scale the pod count up or down. You will also need to supply the **min** number of replicas as well as the **max** number of replicas. These provide a lower and upper bound to ensure minimum capacity and to prevent an excessive number of pods. Service owners are advised to consult within their team to determine reasonable values for these thresholds.

Default values.yaml block

```
# https://github.ol.epicgames.net/charts/epic-app/blob/main/values.yaml
# This configuration is supported by epic-app 2.4.4 and later
epic-app:
  autoscaling:
    enabled: true
    min_replicas: 1
    max_replicas: 2
    cpu:
     enabled: true
     percentage: 60
    # you can uncomment the block below to enable scaling on both memor
    # memory:
     # enabled: true
     # percentage: 60
```

Using the HPA controller for your application can be done manually, but is not recommended, as the autoscaling configuration will be overwritten by the next deploy. Still, if you want to experiment with autoscaling behavior without committing changes, you can use kubectl.

```
kubectl autoscale deployment <your-deployment-name> --cpu-percent=60 --
```

As stated before, when using the epic-app helm chart the default configuration for any application deployed to a Substrate cluster uses the the following directives. (We **recommend** that you change these defaults for your application. At least the max_replicas). In the next section we will discuss how to achieve this.

Example: Configuring an application to use Kubernetes HPA

In this example, we'll configure an application from scratch and configure it to use the HPA autoscaler. This is a very simple application using epic-app to deploy an nginx container.

Create application directory and a blank Chart.yaml file.

```
mkdir -p <application-name>/deploy/helm
cd <application-name>/deploy/helm
touch Chart.yaml
```

Update the Chart.yaml file.

```
apiVersion: v1
description: Example Application
name: example-app
version: 0.1.0

dependencies:
- name: epic-app
  # Check the releases page for latest version.
  # https://github.ol.epicgames.net/charts/epic-app/releases
  version: 2.4.4
  repository: "https://charts.substrate.on.epicgames.com"
```

Before continuing, we'll invoke **helm template** to show the default Kubernetes configuration generated by the epic-app chart.

```
helm dependency build
helm template example-app .
```

The output of the **helm template** shows the Kubernetes manifest that is generated by the chart. There are several objects in the output, but for this example we're focused on the **HorizontalPodAutoscaler** object.

Output of helm template example-app .

```
...
---
# Source: example-app/charts/epic-app/templates/hpa.yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: example-app
  labels:
    helm.sh/chart: epic-app-2.4.3
    app.kubernetes.io/name: epic-app
    app.kubernetes.io/instance: example-app
    app.kubernetes.io/managed-by: Helm
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: example-app
  minReplicas: 1
  maxReplicas: 2
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 60
```

As you can see, by default **epic-app** enables CPU autoscaling with a minimum replica count of 1, max replica count of 2, and a CPU scaling threshold of 60% utilization. Let's change this a bit. This requires creating a **values.dev.yaml** to pass in custom values to the helm chart. Values file names are arbitrary, but we recommend naming your files **values.<env>.yaml** - this allows you to specify different inputs depending on the environment.

Let's adjust the max_replicas and cpu scaling threshold for our deployment. In our example, we use a max_replicas of 125 and a cpu scaling threshold of 75, but the correct value for your app depends on a combination of application-specific factors. We advise you to consult within your team to come up with appropriate values.

We'll also finish up our application deployment by filling out the **containers** section to specify that we want to deploy the nginx container we described earlier.

epic-app values.dev.yaml file

```
# all inputs with this key are passed to the epic-app chart
epic-app:
  autoscaling:
    enabled: true
    cpu:
     enabled: true
     percentage: 75 # changed from the default of 60
    min_replicas: 1
    max_replicas: 125 # changed from the default of 2

  containers:
    example-app:
      image:
         name: nginx
         tag: 1.23.2
      ports:
        - 8080
      resources:
        limits:
          cpu: 500m
        requests:
          cpu: 200m
```

Let's re-run **helm template** to see the new Kubernetes config that will be generated.

```
helm template example-app --values values.dev.yaml .
```

Note the changes below to the **maxReplicas** and **averageUtilization**.

HPA output of helm template command

```
...
---
# Source: example-app/charts/epic-app/templates/hpa.yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: example-app
  labels:
    helm.sh/chart: epic-app-2.4.3
    app.kubernetes.io/name: epic-app
    app.kubernetes.io/instance: example-app
    app.kubernetes.io/managed-by: Helm
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: example-app
  minReplicas: 1
  maxReplicas: 125
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 75
```

Now you can deploy this to your cluster:

```
helm -n <namespace> install example-app --values values.dev.yaml .
```

Once the helm chart has deployed to a Substrate cluster. Run the following kubectl command(s) to inspect the setup.

```
kubectl get hpa -n <namespace>

# you should see similar output as below.

NAME                  REFERENCE                         TARGETS   MINPODS
zero-to-one-lab-02    Deployment/zero-to-one-lab-02    2%/75%    1

kubectl describe hpa zero-to-one-lab-02 -n <namespace>
```

HPA with Kubernetes Manifests

When not using epic-app, you'll need to amend your chart or Kubernetes manifest with a new **HorizontalPodAutoscaler** object. An example is provided below. For more information on configuring these objects directly, see the official Horizontal Pod Autoscaler documentation.

```
# This example is based on the autoscaling/v1 API. This version of the
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: example-app
spec:
  maxReplicas: 2
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: example-app # this indicates that the autoscaling config shou
  targetCPUUtilizationPercentage: 75
status:
  currentReplicas: 0
  desiredReplicas: 0
```

Intro to Kubernetes Event Driven Autoscaling (KEDA)

Additional Context

KEDA is a Kubernetes-based Event Driven Autoscaler.

- started by Red Hat and Microsoft in Feb 2019.
- automatically scale Deployments, Jobs and other Custom resources. (can scale StatefulSets if defined)
- can scale on events in targets. eg. messages in a AWS SQS queue.
- can scale to/from zero
- does not manipulate data, just scales the workload by providing metrics to the Horizontal Pod Autoscaler (HPA)
- provides a fallback replicas count in case of problems
- allows for "tweaking" scaling behavior of HPA

Using KEDA with epic-app

KEDA is configured using ScaledObject as compared with the HPA's HorizontalPodAutoscaler. To help illustrate the differences, we'll configure a CPU-based scaler that functions identically to the HPA scaler we configured above. While KEDA supports over 50 types of scalers, we use the CPU-based scaler to help illustrate the differences between the two options.

Let's update the values.<env>.yaml file created above. and disable **autoscaling:** and enable **keda:**. (*Review the predefined scalers in the default values.yaml for epic-app helm chart*.)

Enable the **cpu** ScaledObject defining the min/max replicas as we did for our HPA config. We define the scaleTargetRef: setting the apiVersion, kind and name. Under the advanced: section we can "tweak" scaling behavior for the HPA created for the workload. Under the **triggers** section we configure the type of KEDA scaler to use and define the required specific metadata for the scaler.

epic-app values.yaml for KEDA CPU scaler

```
epic-app:

  autoscaling:
    enabled: false

    keda:
      enabled: true
      scaled_objects:
        cpu:
          minReplicaCount: 2
          maxReplicaCount: 10
          scaleTargetRef:
            apiVersion: apps/v1
            name: example-app
          # https://keda.sh/docs/2.9/concepts/scaling-deployments/#trig
          triggers:
          - metadata:
              type: Utilization
              value: "75"
            type: cpu
```

Static Manifest KEDA CPU scaler if not using epic-app chart.

Controlling HPA scaling behavior via KEDA

There may be times when you need to manage scaling behavior, such as how many pods are created or removed at once, and the interval between pod count changes. This is achieved by added an **advanced**: section to the ScaledObjects spec.

epic-app values.yaml for KEDA CPU scaler

```
epic-app:

  autoscaling:
    enabled: false

    keda:
      enabled: true
      scaled_objects:
        cpu:
          minReplicaCount: 2
          maxReplicaCount: 10
          scaleTargetRef:
            apiVersion: apps/v1
            kind: Deployment
            name: example-app
          # https://keda.sh/docs/2.9/concepts/scaling-deployments/#adva
          # Alter scaling behavior
          advanced:
            horizontalPodAutoscalerConfig:
              behavior:
                scaleDown:
                  policies:
                  - periodSeconds: 60
                    type: Pods
                    value: 5
                  stabilizationWindowSeconds: 300
                scaleUp:
                  policies:
                  - periodSeconds: 60
                    type: Pods
                    value: 10
                  stabilizationWindowSeconds: 0
            # WARNING: this setting controls what happens if the KEDA S
            # size (before scaling occurred). This can be disastrous if
            # When set to false, removal of the scaler will leave the d
            restoreToOriginalReplicaCount: false
          # https://keda.sh/docs/2.9/concepts/scaling-deployments/#trig
          triggers:
          - metadata:
              type: Utilization
              value: "75"
            type: cpu
```

Static Manifest for KEDA CPU scaler managing the scaling behavior when not using epic-app chart.

Once the helm chart is deployed to your cluster. Run:

NOTE: since the KEDA operator uses HPA to manage the scaling. The name of the HPA object created will be prepended with keda-hpa

```
kubectl get hpa -n <namespace>

# you should see similar output as below.
NAME                               REFERENCE                          TARGE
keda-hpa-zero-to-one-lab-02-cpu    Deployment/zero-to-one-lab-02      2%/75
```

```
kubectl get scaledobject -n <namespace>
```

With the introduction of KEDA into the Substrate environment, the platform extended the scaling options by adding event triggers that scale up/down your application. Out of the box KEDA supports 50+ scalers, including CPU, memory, Apache Kafka, AWS DynamoDB, AWS SQS Queue and many more listed here. Within Substrate outside of the CPU scaler we've see the SQS scaler used to scale applications. This section of the documentation will describe how to implement that pattern.

Using KEDA with Kubernetes Manifests

If using your own chart or raw Kubernetes manifests, you can still use KEDA, though you will need to configure more details than with epic-app. See the documentation for ScaledObject for more information.

Below is a ScaledObject configuration representing simple CPU scaling example configuration used earlier in this doc.

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: example-app-cpu
spec:
  maxReplicaCount: 10
  minReplicaCount: 2
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: example-app
  triggers:
  - metadata:
      type: Utilization
      value: "75"
    type: cpu
```

This config expands on the sample CPU scaling configuration with some more advanced options around scaling behavior.

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: example-app-cpu
spec:
    advanced:
      horizontalPodAutoscalerConfig:
        behavior:
          scaleDown:
            policies:
            - periodSeconds: 60
              type: Pods
              value: 5
            stabilizationWindowSeconds: 300
          scaleUp:
            policies:
            - periodSeconds: 60
              type: Pods
              value: 10
            stabilizationWindowSeconds: 0
    maxReplicaCount: 10
    minReplicaCount: 2
    scaleTargetRef:
      apiVersion: apps/v1
      kind: Deployment
      name: example-app
    triggers:
    - metadata:
        type: Utilization
        value: "75"
      type: cpu
```

Advanced Examples With KEDA Scalers

KEDA supports over 50 scaler types, which is too many to provide practical examples of everything. However, it's worth illustrating with one example so that you can see the utility of KEDA for more complex scaling scenarios. Below we'll cover how to scale a configuration based on the number of messages in an SQS queue.

AWS SQS Queue

Creating a AWS SQS Queue

For demo purposes we'll create a new SQS queue. Below is an example terraform that will create it. For more information on how to use Terraform at Epic, see the following docs:

- Using Terraform With Substrate (for manually-driven workflows)
- TFE User Guide (for automated workflows with Terraform Enterprise)

```
# https://registry.terraform.io/providers/hashicorp/aws/latest/docs/res
resource "aws_sqs_queue" "example_app" {
  name                       = "example-app-s3-events-queue"
  delay_seconds              = 0
  max_message_size           = 262144 # 256Kb
  message_retention_seconds = 345600 # 4 days
  receive_wait_time_seconds = 0

  lifecycle {
    prevent_destroy = true
  }

  tags = {
    "epic/substrate/application" = "example-app"
    "epic/substrate/account-id"  = "beef"
    "epic/substrate/region"      = "use1a"
    "epic/substrate/tier"        = "live"
  }
}
```

NOTE: The IAM permissions for accessing the SQS from KEDA deployed in your cluster is already in place. As a result you don't have to worry about providing access.

Update applications helm values file

Once the SQS queue has been created the next step is creating the KEDA scaledObject. This can be achieved using the epic-app helm chart.

example: deploy/helm/values.<env>.yaml for KEDA sqs scaller

```
epic-app:

  autoscaling:
    enabled: false

    keda:
      enabled: true
      scaled_objects:
        sqs:
          minReplicaCount: 1
          maxReplicaCount: 10
          scaleTargetRef:
            apiVersion: apps/v1
            kind: Deployment
            name: example-app
          # https://keda.sh/docs/2.9/concepts/scaling-deployments/#trig
          triggers:
          - metadata:
              type: aws-sqs-queue
                awsRegion: us-east-1
                identityOwner: operator
                queueLength: "50"
                queueURL: https://sqs.us-east-1.amazonaws.com/967231986
```

Alternatively if you are using your own chart or raw Kubernetes manifests, you can use this ScaledObject configuration:

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: example-app-sqs
spec:
    maxReplicaCount: 10
    minReplicaCount: 1
    scaleTargetRef:
      apiVersion: apps/v1
      kind: Deployment
      name: zero-to-one-lab-10
    triggers:
    - metadata:
        awsRegion: us-east-1
        identityOwner: operator
        queueLength: "50"
        queueURL: https://sqs.us-east-1.amazonaws.com/967231986811/exam
      type: aws-sqs-queue
```

NOTE the following directives:

- **queueLength** value. Here we are saying that a single pod can handle 50 items in the queue. Which means if there is 200 messages, the scaler scales to 4 pods.
- **identityOwner** - defines who has permissions on the SQS Queue. The Pod Identity or the KEDA operator. Within a Substrate cluster the KEDA operator has the permissions. Via a global IAM role named: aws-svc-global-k8s-argocd-keda

Working with KEDA Scalers That Require Authentication

Due to the wide range of supported scalers, the one selected may require credentials to read the metrics value. Within Substrate, the credentials can be pulled from Substrate Vault by the KEDA operator. To achieve this there are two authentication patterns supported by KEDA

- TriggerAuthentication (TA) per-namespace credentials
- ClusterTriggerAuthentication (CTA) global credentials

Each of the stated methods use support Kubernetes secrets, Hashicorp Vault, and a number of other patterns. Within Substrate we recommend using a Hashicorp vault secret. To achieve this

1. Create the vault path and store the secret if it doesn't already exists.
2. Share the vault path with the `<account>-<env>.<account>-<env>-<clustername>.epic-system.keda-operator` via SSSM
3. Implement TriggerAuthentication custom resource.

Create TriggerAuthentication

Updated values.yaml file enabling the trigger_authentications

```
epic-app:

  autoscaling:
    enabled: false

    keda:
      # https://keda.sh/docs/2.9/concepts/authentication/#hashicorp-vau
      trigger_authentications:  # https://keda.sh/docs/2.6/concepts/aut
        redis-auth-secret:
          hashiCorpVault:
            address: https://vault.substrate.on.epicgames.com
            authentication: kubernetes
            role: ffce-dev.ffce-dev-substrate-labs.epic-system.keda-ope
            mount: kubernetes/ffce-dev/ffce-dev-substrate-labs
            credential:
              serviceAccount: /var/run/secrets/kubernetes.io/serviceacc
            secrets:
            - parameter: password
```

```
                path: /secret/data/<brand>/<project>/use1a/dev/runtime/re
                key: password
```

Reference TriggerAuthentication

To use the above TriggerAuthentication object. Update your values.<env>.yaml

```
epic-app:

  autoscaling:
    enabled: false

    keda:
      redis:
      ...
        # https://keda.sh/docs/2.9/concepts/scaling-deployments/#trigge
        triggers:
         - type: redis
           metadata:
             address: redis.lab-student-djrtggat.svc.cluster.local:6379
             listName: slabs
             listLength: "2"
           authenticationRef:
             name: zero-to-one-lab-10-redis-auth-secret
```

Manually Scaling Pod Deployments

While not specific to autoscaling, it's worth mentioning that pod deployments can be scaled manually if needed. You might do this, for example, in situations where you haven't configured autoscaling and need to adjust capacity in a pinch, such as ahead of a large event.

```
kubectl create deployment example-app --image=nginx:1.23.1 -n <namespac
kubectl rollout status deployment example-app
kubectl scale deployment example-app -n <namespace> --replicas=5
kubectl get pods -n <namespace>
```

Next steps

Up until now we've discussed using the epic-app helm chart for autoscaling. Starting with the Horizontal Pod Autoscaler (HPA) CPU metrics to scale our Deployment. Then switching to using KEDA and letting it provide the metrics for HPA to scale based on CPU. We then discuss using the number of items in an AWS SQS queue to scale our application. Finally we discussed using Substrate Vault with KEDA's TriggerAuthentication custom resource to manage the secret needed by scaler you may use where credentials hadn't been provided.

The next steps in this journey is identifying which of the 50+ scaler to use to scale your workloads. Start that journey by reviewing our KEDA documentation section.

Tips

1. When migrating from HPA to KEDA, we recommend using the same values used for the HPA's minReplicas and maxReplicas being "migrated" from.
2. When setting the maxRelicaCount consider the number of pods need when we're handling an event with expected high CCU (KEDA defaults to 100).
3. The node autoscaler Karpenter will add additional an node(s) if pod(s) end up in a unschedulable stated due to a HPA or KEDA scaling event when setting the max to handle high CCU counts.
4. KEDA **defaults to zero** replicas, so always set the minReplicaCount: 1 if your workload isn't responsible for processing a message in a queue.
5. Configure Fallback(s) for the selected scaler.
6. Within the advanced: section the ability to configure **horizontalPodAutoscalerConfig.behavior**. For example the stabilizationWindowSeconds directive "is used to restrict the flapping of replicas count when the metrics used for scaling keep fluctuating."

7. Each cluster has a KEDA Workload Utilization dashboard. To locate the dashboard for your team's cluster. Within NewRelic UI navigate to **Dashboards → Type: epic-system-dashoard-ffce-dev**

Additional resources

- epic-app helm chart
- Scaling Deployments, StatefulSets & Custom Resources
- CPU
- AWS SQS
- Authentication

Page Information:
Page ID: 81068324
Space: Cloud Developer Platform
Downloaded: 2025-07-12 04:09:05