Golang-based operator is the etcd operator for managing an etcd key-value store and automating operational tasks like backing up and restoring the database.

If you are looking for an operator written in the Java programming language, the *Strimzi Operator* is an excellent example of an operator that manages a complex messaging system like Apache Kafka on Kubernetes. Another good starting point for Java-based operators is the *Java Operator Plugin*, part of the Operator-SDK. As of 2023, it is still a young initiative; the best entry point for learning more about creating Java-based operators is the tutorial that explains the process to create a fully working operator.

Discussion

While we have learned how to extend the Kubernetes platform, operators are still not a silver bullet. Before using an operator, you should carefully look at your use case to determine whether it fits the Kubernetes paradigm.

In many cases, a plain controller working with standard resources is good enough. This approach has the advantage that it doesn't need any cluster-admin permission to register a CRD, but it has its limitations when it comes to security and validation.

An operator is a good fit for modeling a custom domain logic that fits nicely with the declarative Kubernetes way of handling resources with reactive controllers.

More specifically, consider using an operator with CRDs for your application domain for any of the following situations:

- You want tight integration into the already-existing Kubernetes tooling like `kubectl`.
- You are working on a greenfield project where you can design the application from the ground up.
- You benefit from Kubernetes concepts like resource paths, API groups, API versioning, and especially namespaces.
- You want to have good client support for accessing the API with watches, authentication, role-based authorization, and selectors for metadata.

If your custom use case fits these criteria, but you need more flexibility in how custom resources can be implemented and persisted, consider using a custom API Server. However, you should also not consider Kubernetes extension points as the golden hammer for everything.

If your use case is not declarative, if the data to manage does not fit into the Kubernetes resource model, or you don't need a tight integration into the platform, you are probably better off writing your standalone API and exposing it with a classical Service or Ingress object.

The Kubernetes documentation itself also has a chapter for suggestions on when to use a controller, operator, API aggregation, or custom API implementation.

More Information

- Operator Example
- OpenAPI V3
- Kubebuilder
- Operator Framework
- Metacontroller
- Client Libraries
- Extend the Kubernetes API with CustomResourceDefinitions
- Custom Resources
- Sample-Controller
- What Are Red Hat OpenShift Operators?

CHAPTER 29

Elastic Scale

The *Elastic Scale* pattern covers application scaling in multiple dimensions: horizontal scaling by adapting the number of Pod replicas, vertical scaling by adapting resource requirements for Pods, and scaling the cluster itself by changing the number of cluster nodes. While all of these actions can be performed manually, in this chapter we explore how Kubernetes can perform scaling based on load automatically.

Problem

Kubernetes automates the orchestration and management of distributed applications composed of a large number of immutable containers by maintaining their declaratively expressed desired state. However, with the seasonal nature of many workloads that often change over time, it is not an easy task to figure out how the desired state should look. Accurately identifying how many resources a container will require and how many replicas a service will need at a given time to meet service-level agreements takes time and effort. Luckily, Kubernetes makes it easy to alter the resources of a container, the desired replicas for a service, or the number of nodes in the cluster. Such changes can happen either manually, or given specific rules, can be performed in a fully automated manner.

Kubernetes not only can preserve a fixed Pod and cluster setup but can also monitor external load and capacity-related events, analyze the current state, and scale itself for the desired performance. This kind of observation is a way for Kubernetes to adapt and gain antifragile traits based on actual usage metrics rather than anticipated factors. Let's explore the different ways we can achieve such behavior and how to combine the various scaling methods for an even greater experience.

Solution

There are two main approaches to scaling any application: horizontal and vertical. *Horizontally* in the Kubernetes world equates to creating more replicas of a Pod. *Vertically* scaling implies giving more resources to running containers managed by Pods. While it may seem straightforward on paper, creating an application configuration for autoscaling on a shared cloud platform without affecting other services and the cluster itself requires significant trial and error. As always, Kubernetes provides a variety of features and techniques to find the best setup for our applications, and we explore them briefly here.

Manual Horizontal Scaling

The manual scaling approach, as the name suggests, is based on a human operator issuing commands to Kubernetes. This approach can be used in the absence of autoscaling or for gradual discovery and tuning of the optimal configuration of an application matching the slow-changing load over long periods. An advantage of the manual approach is that it also allows anticipatory rather than reactive-only changes: knowing the seasonality and the expected application load, you can scale it out in advance, rather than reacting to an already-increased load through autoscaling, for example. We can perform manual scaling in two styles.

Imperative scaling

A controller such as ReplicaSet is responsible for making sure a specific number of Pod instances are always up and running. Thus, scaling a Pod is as trivially simple as changing the number of desired replicas. Given a Deployment named `random-generator`, scaling it to four instances can be done in one command, as shown in Example 29-1.

Example 29-1. Scaling a Deployment's replicas on the command line

```
kubectl scale random-generator --replicas=4
```

After such a change, the ReplicaSet could either create additional Pods to scale up or, if there are more Pods than desired, delete them to scale down.

Declarative scaling

While using the scale command is trivially simple and good for quick reactions to emergencies, it does not preserve this configuration outside the cluster. Typically, all Kubernetes applications would have their resource definitions stored in a source control system that also includes the number of replicas. Recreating the ReplicaSet from its original definition would change the number of replicas back to its previous number. To avoid such a configuration drift and to introduce operational processes for backporting changes, it is a better practice to change the desired number of replicas declaratively in the ReplicaSet or some other definition and apply the changes to Kubernetes, as shown in Example 29-2.

Example 29-2. Using a Deployment for declaratively setting the number of replicas

```
kubectl apply -f random-generator-deployment.yaml
```

We can scale resources managing multiple Pods such as ReplicaSets, Deployments, and StatefulSets. Notice the asymmetric behavior in scaling a StatefulSet with persistent storage. As described in Chapter 12, "Stateful Service", if the StatefulSet has a `.spec.volumeClaimTemplates` element, it will create PVCs while scaling, but it won't delete them when scaling down to preserve the storage from deletion.

Another Kubernetes resource that can be scaled but follows a different naming convention is the Job resource, which we described in Chapter 7, "Batch Job". A Job can be scaled to execute multiple instances of the same Pod at the same time by changing the `.spec.parallelism` field rather than `.spec.replicas`. However, the semantic effect is the same: increased capacity with more processing units that act as a single logical unit.

For describing resource fields, we use a JSON path notation. For example, `.spec.replicas` points to the `replicas` field of the resource's `spec` section.

Both manual scaling styles (imperative and declarative) expect a human to observe or anticipate a change in the application load, make a decision on how much to scale, and apply it to the cluster. They have the same effect, but they are not suitable for dynamic workload patterns that change often and require continuous adaptation. Next, let's see how we can automate scaling decisions themselves.

Horizontal Pod Autoscaling

Many workloads have a dynamic nature that varies over time and makes it hard to have a fixed scaling configuration. But cloud native technologies such as Kubernetes enable you to create applications that adapt to changing loads. Autoscaling in Kubernetes allows us to define a varying application capacity that is not fixed but instead ensures just enough capacity to handle a different load. The most straightforward approach to achieving such behavior is by using a HorizontalPodAutoscaler (HPA) to horizontally scale the number of Pods. HPA is an intrinsic part of Kubernetes and does not require any extra installation steps. One important limitation of the HPA is that it can't scale down to zero Pods so that no resources are consumed at all if nobody is using the deployed workload. Luckily, Kubernetes add-ons offer scale-to-zero and transform Kubernetes into a true serverless platform. Knative and KEDA are the most prominent of such Kubernetes extensions. We will have a look at both in "Knative" on page 317 and "KEDA" on page 321, but let's first see how Kubernetes offers horizontal autoscaling out of the box.

Kubernetes HorizontalPodAutoscaler

The HPA is best explained with an example. An HPA for the `random-generator` Deployment can be created with the command in Example 29-3. For the HPA to have any effect, it is important that the Deployment declare a `.spec.resources.requests` limit for the CPU as described in Chapter 2, "Predictable Demands". Another requirement is enabling the metrics server, which is a cluster-wide aggregator of resource usage data.

Example 29-3. Create HPA definition on the command line

```
kubectl autoscale deployment random-generator --cpu-percent=50 --min=1 --max=5
```

The preceding command will create the HPA definition shown in Example 29-4.

Example 29-4. HPA definition

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: random-generator
spec:
  minReplicas: 1                ❶
  maxReplicas: 5                ❷
  scaleTargetRef:               ❸
    apiVersion: apps/v1
    kind: Deployment
    name: random-generator
  metrics:
  - resource:
      name: cpu
      target:
        averageUtilization: 50 ❹
        type: Utilization
    type: Resource
```

❶ Minimum number of Pods that should always run.

❷ Maximum number of Pods until the HPA can scale up.

❸ Reference to the object that should be associated with this HPA.

❹ Desired CPU usage as a percentage of the Pods' requested CPU resource. For example, when the Pods have a `.spec.resources.requests.cpu` of 200m, a scale-up happens when on average more than 100m CPU (= 50%) is utilized.

This definition instructs the HPA controller to keep between one and five Pod instances to retain an average Pod CPU usage of around 50% of the specified CPU resource limit in the Pod's `.spec.resources.requests` declaration. While it is possible to apply such an HPA to any resource that supports the `scale` subresource such as Deployments, ReplicaSets, and StatefulSets, you must consider the side effects. Deployments create new ReplicaSets during updates but without copying over any HPA definitions. If you apply an HPA to a ReplicaSet managed by a Deployment, it is not copied over to new ReplicaSets and will be lost. A better technique is to apply the HPA to the higher-level Deployment abstraction, which preserves and applies the HPA to the new ReplicaSet versions.

Now, let's see how an HPA can replace a human operator to ensure autoscaling. At a high level, the HPA controller performs the following steps continuously:

1. It retrieves metrics about the Pods that are subject to scaling according to the HPA definition. Metrics are not read directly from the Pods but from the Kubernetes Metrics APIs that serve aggregated metrics (and even custom and external metrics if configured to do so). Pod-level resource metrics are obtained from the Metrics API, and all other metrics are retrieved from the Custom Metrics API of Kubernetes.
2. It calculates the required number of replicas based on the current metric value and targeting the desired metric value. Here is a simplified version of the formula:

$$desiredReplicas = \left\lceil currentReplicas \times \frac{currentMetricValue}{desiredMetricValue} \right\rceil$$

For example, if there is a single Pod with a current CPU usage metric value of 90% of the specified CPU resource request value,[1] and the desired value is 50%, the number of replicas will be doubled, as $\left\lceil 1 \times \frac{90}{50} \right\rceil = 2$. The actual implementation is more complicated as it has to consider multiple running Pod instances, cover multiple metric types, and account for many corner cases and fluctuating values as well. If multiple

1 For multiple running Pods, the average CPU utilization is used as *currentMetricValue*.

metrics are specified, for example, then the HPA evaluates each metric separately and proposes a value that is the largest of all. After all the calculations, the final output is a single-integer number representing the number of desired replicas that keep the measured value below the desired threshold value.

The `replicas` field of the autoscaled resource will be updated with this calculated number, and other controllers do their bit of work in achieving and keeping the new desired state. Figure 29-1 shows how the HPA works: monitoring metrics and changing declared replicas accordingly.

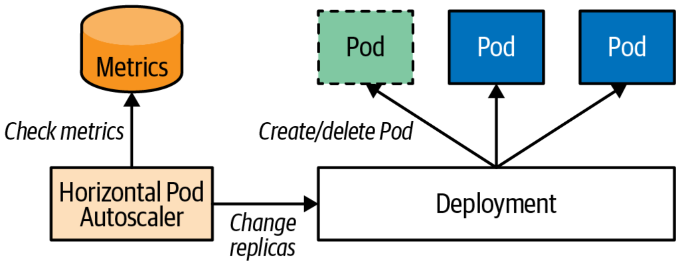

Figure 29-1. Horizontal Pod autoscaling mechanism

Autoscaling is an area of Kubernetes with many low-level details, and each one can have a significant impact on the overall behavior of autoscaling. As such, it is beyond the scope of this book to cover all the details, but "More Information" on page 333 provides the latest up-to-date information on the subject.

Broadly, there are the following metric types:

Standard metrics
: These metrics are declared with `.spec.metrics.resource[].type` equal to `Resource` and represent resource usage metrics such as CPU and memory. They are generic and available for any container on any cluster under the same name. You can specify them as a percentage, as we did in the preceding example, or as an absolute value. In both cases, the values are based on the guaranteed resource amount, which are the container resource `requests` values and not the `limits` values. These are the easiest-to-use metric types generally provided by the metrics server component, which can be launched as cluster add-ons.

Custom metrics
: These metrics with `.spec.metrics.resource[].type` equal to `Object` or `Pod` require a more advanced cluster-monitoring setup, which can vary from cluster to cluster. A custom metric with the Pod type, as the name suggests, describes a Pod-specific metric, whereas the Object type can describe any other object. The custom metrics are served in an aggregated API Server under the

`custom.metrics.k8s.io` API path and are provided by different metrics adapters, such as Prometheus, Datadog, Microsoft Azure, or Google Stackdriver.

External metrics
: This category is for metrics that describe resources that are not a part of the Kubernetes cluster. For example, you may have a Pod that consumes messages from a cloud-based queueing service. In such a scenario, you'll want to scale the number of consumer Pods based on the queue depth. Such a metric would be populated by an external metrics plugin similar to custom metrics. Only one external metrics endpoint can be hooked into the Kubernetes API server. For using metrics from many different external systems, an extra aggregation layer like KEDA is required (see "KEDA" on page 321).

Getting autoscaling right is not easy and involves a little experimenting and tuning. The following are a few of the main areas to consider when setting up an HPA:

Metric selection
: Probably one of the most critical decisions around autoscaling is which metrics to use. For an HPA to be useful, there must be a direct correlation between the metric value and the number of Pod replicas. For example, if the chosen metric is of the Queries-per-Second kind (such as HTTP requests per second), increasing the number of Pods causes the average number of queries to go down as the queries are dispatched to more Pods. The same is true if the metric is CPU usage, as there is a direct correlation between the query rate and CPU usage (an increased number of queries would result in increased CPU usage). For other metrics such as memory consumption, that is not the case. The issue with memory is that if a service consumes a certain amount of memory, starting more Pod instances most likely will not result in a memory decrease unless the application is clustered and aware of the other instances and has mechanisms to distribute and release its memory. If the memory is not released and reflected in the metrics, the HPA would create more and more Pods in an effort to decrease it, until it reaches the upper replica threshold, which is probably not the desired behavior. So choose a metric that is directly (preferably linearly) correlated to the number of Pods.

Preventing thrashing
: The HPA applies various techniques to avoid rapid execution of conflicting decisions that can lead to a fluctuating number of replicas when the load is not stable. For example, during scale-up, the HPA disregards high CPU usage samples when a Pod is initializing, ensuring a smoothing reaction to increasing load. During scale-down, to avoid scaling down in response to a short dip in usage, the controller considers all scale recommendations during a configurable time window and chooses the highest recommendation from within the window. All this makes the HPA more stable when dealing with random metric fluctuations.

Delayed reaction

Triggering a scaling action based on a metric value is a multistep process involving multiple Kubernetes components. First, it is the cAdvisor (container advisor) agent that collects metrics at regular intervals for the Kubelet. Then the metrics server collects metrics from the Kubelet at regular intervals. The HPA controller loop also runs periodically and analyzes the collected metrics. The HPA scaling formula introduces some delayed reaction to prevent fluctuations/thrashing (as explained in the previous point). All this activity accumulates into a delay between the cause and the scaling reaction. Tuning these parameters by introducing more delay makes the HPA less responsive, but reducing the delays increases the load on the platform and increases thrashing. Configuring Kubernetes to balance resources and performance is an ongoing learning process.

Tuning the autoscale algorithm for the HPA in Kubernetes can be complex. To help with this, Kubernetes provides the `.spec.behavior` field in the HPA specification. This field allows you to customize the behavior of the HPA when scaling the number of replicas in a Deployment.

For each scaling direction (up or down), you can use the `.spec.behavior` field to specify the following parameters:

`policies`

These describe the maximum number of replicas to scale in a given period.

`stabilizationWindowSeconds`

This specifies when the HPA will not make any further scaling decisions. Setting this field can help to prevent thrashing effects, where the HPA rapidly scales the number of replicas up and down.

Example 29-5 shows how the behavior can be configured. All behavior parameters can also be configured on the CLI with `kubectl autoscale`.

Example 29-5. Configuration of the autoscaling algorithm

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
...
spec:
  ...
  behavior:
    scaleDown:                        ❶
      stabilizationWindowSeconds: 300 ❷
      policies:
      - type: Percent                 ❸
        value: 10
        periodSeconds: 60
    scaleUp:                          ❹
      policies:
      - type: Pods                    ❺
        value: 4
        periodSeconds: 15
```

❶ Scaling behavior when scaling down.

❷ A 5-minute minimum window for down-scaling decisions to prevent flapping.

❸ Scale down at most 10% of the current replicas in one minute.

❹ Scaling behavior when scaling up.

❺ Scale up at most four Pods within 15 seconds.

Please refer to the Kubernetes documentation on configuring the scaling behavior for all the details and usage examples.

While the HPA is very powerful and covers the basic needs for autoscaling, it lacks one crucial feature: scale-to-zero for stopping all Pods of an application if it is not used. That's important so that it does not cause any costs based on memory, CPU, or network usage. However, scaling to zero is not so hard; the tricky part is waking up again and scaling to at least one Pod by a trigger, like an incoming HTTP request or an event to process.

The following two sections introduce the two most prominent Kubernetes-based add-ons for enabling scale-to-zero: Knative and KEDA. It is essential to understand that Knative and KEDA are not alternative but complementary solutions. Both projects cover different use cases and can ideally be used together. As we will see, Knative specializes in stateless HTTP applications and offers an autoscaling algorithm that goes beyond the capabilities of the HPA. On the other hand, KEDA is a pull-based approach that can be triggered by many different sources, like messages in a Kafka topic or IBM MQ queue.

Let's have a closer look at Knative and KEDA.

Knative

Knative is a CNCF project initiated by Google in 2018, with broad industry support from vendors like IBM, VMware, and Red Hat. This Kubernetes add-on consists of three parts:

Knative Serving
: This is a simplified application deployment model with sophisticated autoscaling and traffic-splitting capabilities, including scale-to-zero.

Knative Eventing
: This provides everything needed to create an Event Mesh to connect event sources that produce CloudEvents[2] with a sink that consumes these events. Those sinks are typically Knative Serving services.

Knative Functions
: This is for scaffolding and building Knative Serving services from source code. It supports various programming languages and offers an AWS Lambda-like programming model.

In this section, we will focus on Knative Serving and its autoscaler for an application that uses HTTP to offer its services. For those workloads, CPU and memory are metrics that only indirectly correlate to actual usage. A much better metric is the number of *concurrent requests* per Pod—i.e., requests that are processed in parallel.

Another HTTP-based metric that Knative can use is *requests per second* (rps). Still, this metric does not say anything about the costs of a single request, so concurrent requests are typically the much better metric to use, as they capture the frequency of requests and the duration of those requests. You can select the scale metric individually for each application or as a global default.

Basing the autoscaling decision on concurrent requests gives a much better correlation to the latency of HTTP request processing than scaling based on CPU or memory consumption can provide.

Historically, Knative used to be implemented as a custom metric adapter for the HPA in Kubernetes. However, it later developed its own implementation in order to have more flexibility in influencing the scaling algorithm and to avoid the bottleneck of being able to register only a single custom metric adapter in a Kubernetes cluster.

While Knative still supports using the HPA for scaling based on memory or CPU usage, it now focuses on using its own autoscaling implementation, called the Knative Pod Autoscaler (KPA). This allows Knative to have more control over the scaling algorithm and to better optimize it for the needs of the application.

The architecture of the KPA is shown in Figure 29-2.

2 CloudEvents is a CNCF standard that describes the format and metadata for events in a cloud context.

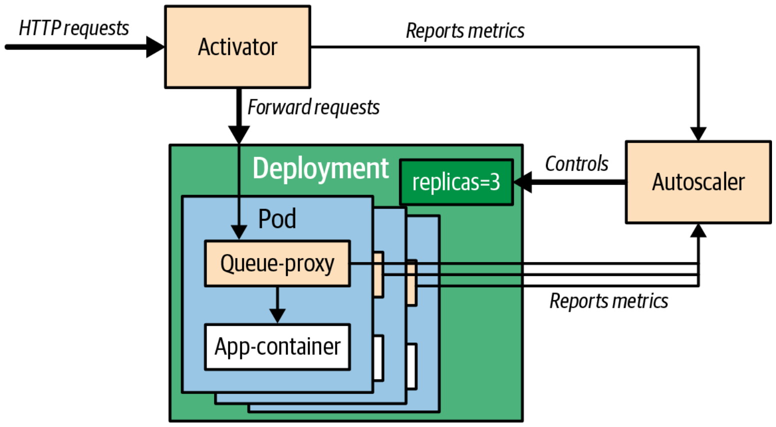

Figure 29-2. Knative Pod Autoscaler

Three components are playing together for autoscaling a service:

Activator
: This is a proxy in front of the application that is always available, even when the application is scaled down to zero Pods. When the application is scaled down to zero, and a first request comes in, the request gets buffered, and the application is scaled up to at least one Pod. It's important to note that during a *cold start*, all incoming requests will be buffered to ensure that no requests are lost.

Queue proxy
: The queue proxy is an ambassador sidecar described in Chapter 18 that is injected into the application's Pod by the Knative controller. It intercepts the request path for collecting metrics relevant to autoscaling, like concurrent requests.

Autoscaler
: This is a service running in the background that is responsible for the scaling decision based on the data it gets from the activator and queue-proxy. The autoscaler is the one that sets the replica count in the application's ReplicaSet.

The KPA algorithm can be configured in many ways to optimize the autoscaling behavior for any workload and traffic shape. Table 29-1 shows some of the configuration options for tuning the KPA for individual services via annotations. Similar configuration options also exist for global defaults that are stored in a ConfigMap. You can find the full set of all autoscaling configuration options in the Knative documentation. This documentation has more details about the Knative scaling algorithm, like dealing with bursty workloads by scaling up more aggressively when the increase in concurrent requests is over a threshold.

Table 29-1. Important Knative scaling parameters. `autoscaling.knative.dev/`, the common annotation prefix, has been omitted.

Annotation	Description	Default
`target`	Number of simultaneous requests that can be processed by each replica. This is a soft limit and might be temporarily exceeded in case of a traffic burst. `.spec.concurrencyLimit` is used as a hard limit that can't be crossed.	100
`target-utilization-percentage`	Start creating new replicas if this fraction of the concurrency limit has been reached.	70
`min-scale`	Minimum number of replicas to keep. If set to a value greater than zero, the application will never scale down to zero.	0
`max-scale`	Upper bound for the number of replicas; zero means unlimited scaling.	0
`activation-scale`	How many replicas to create when scaling up from zero.	1
`scale-down-delay`	How long scale-down conditions must hold before scaling down. Useful for keeping replicas warm before scaling zero in order to avoid cold start time.	0s
`window`	Length of the time window over which metrics are averaged to provide the input for scaling decisions.	60s

Example 29-6 shows a Knative service that deploys an example application. It looks similar to a Kubernetes Deployment. However, behind the scenes, the Knative operator creates the Kubernetes resources needed to expose your application as a web service, i.e., a ReplicaSet, Kubernetes Service, and Ingress for exposing the application to the outside of your cluster.

Example 29-6. Knative service

```
apiVersion: serving.knative.dev/v1                      ❶
kind: Service
metadata:
  name: random
  annotations:
    autoscaling.knative.dev/target: "80"                ❷
    autoscaling.knative.dev/window: "120s"
spec:
  template:
    spec:
      containers:
      - image: k8spatterns/random                       ❸
```

❶ Knative also uses Service for the resource name but with the API group `serving.knative.dev`, which is different from a Kubernetes Service from the `core` API group.

❷ Options for tuning the autoscaling algorithm. See Table 29-1 for the available options.

❸ The only mandatory argument for a Knative Service is a reference to a container image.

We only briefly touch on Knative here. There is much more that can help you in operating the Knative autoscaler. Please check out the online documentation for more features of Knative Serving, like traffic splitting for the complex rollout scenarios we described in Chapter 3, "Declarative Deployment". Also, if you are following an event-driven architecture (EDA) paradigm for your applications, Knative Eventing and Knative Functions have a lot to offer.

KEDA

Kubernetes Event-Driven Autoscaling (KEDA) is the other important Kubernetes-based autoscaling platform that supports scale-to-zero but has a different scope than Knative. While Knative supports autoscaling based on HTTP traffic, KEDA is a pull-based approach that scales based on external metrics from different systems. Knative and KEDA play very well together, and there is only a little overlap,[3] so nothing prevents you from using both add-ons together.

So, what is KEDA? KEDA is a CNCF project that Microsoft and Red Hat created in 2019 and consists of the following components:

- The KEDA Operator reconciles a ScaledObject custom resource that connects the scaled target (e.g., a Deployment or StatefulSet) with an autoscale trigger that connects to an external system via a so-called *scaler*. It is also responsible for configuring the HPA with the external metrics service provided by KEDA.
- KEDA's metrics service is registered as an APIService resource in the Kubernetes API aggregation layer so that the HPA can use it as an external metrics service.

Figure 29-3 illustrates the relationship between the KEDA Operator, metrics service, and the Kubernetes HPA.

3 KEDA initially did not support HTTP-triggered autoscaling, and although there is now a KEDA HTTP add-on it is still in its infancy (in 2023), requires a complex setup, and would need to catch up quite a bit to reach the maturity of the KPA that is included out of the box in Knative.

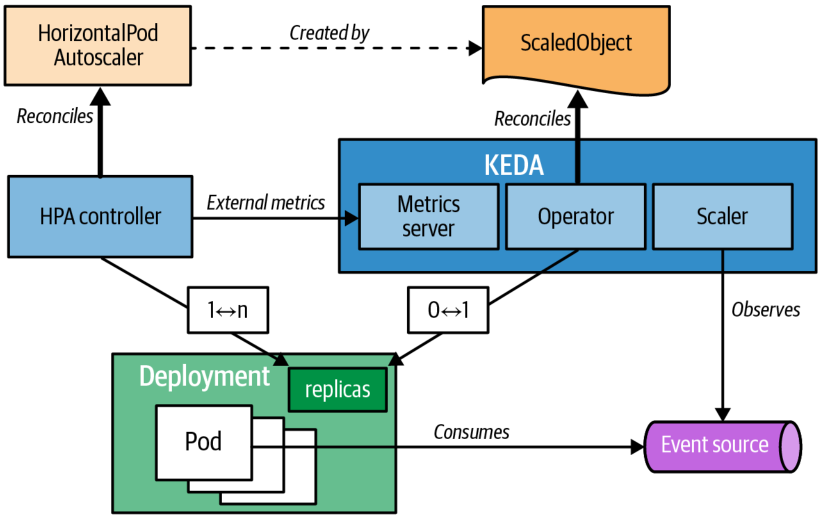

Figure 29-3. KEDA autoscaling components

While Knative is a complete solution that completely replaces HPA for a consumption-based autoscaling, KEDA is a hybrid solution. KEDA's autoscaling algorithm distinguishes between two scenarios:

- Activation by scaling from zero replicas to one (*0 ↔ 1*): This action is performed by the KEDA operator itself when it detects that a used scaler's metric exceeds a certain threshold.
- Scaling up and down when running (*1 ↔ n*): When the workload is already active, the HPA takes over and scales based on the external metric that KEDA offers.

The central element for KEDA is the custom resource ScaledObject, provided by the user to configure KEDA-based autoscaling and playing a similar role as the HorizontalPodAutoscaler resource. As soon as the KEDA operator detects a new instance of ScaledObject, it automatically creates a HorizontalPodAutoscaler resource that uses the KEDA metrics service as an external metrics provider and the scaling parameters.

Example 29-7 shows how you can scale a Deployment based on the number of messages in an Apache Kafka topic.

Example 29-7. ScaledObject definition

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: kafka-scaledobject
spec:
  scaleTargetRef:
    name: kafka-consumer                          ❶
  pollingInterval: 30                             ❷
  triggers:
    - type: kafka                                 ❸
      metadata:
        bootstrapServers: bootstrap.kafka.svc:9092 ❹
        consumerGroup: my-group
        topic: my-topic
```

❶ Reference to a Deployment with the name `kafka-consumer` that should be autoscaled. You can also specify other scalable workloads here; Deployment is the default.

❷ In the action phase (scale from zero), poll every 30 seconds for the metric value. In this example, it is the number of messages in a Kafka topic.

❸ Select the Apache Kafka scaler.

❹ Configuration options for the Apache Kafka scaler—i.e., how to connect to the Kafka cluster and which topic to monitor.

KEDA provides many out-of-the-box scalers that can be selected to connect to external systems for the autoscaling stimulus. You can obtain the complete list of directly supported scalers from the KEDA home page. In addition, you can easily integrate custom scalers by providing an external service that communicates with KEDA over a gRPC-based API.

KEDA is a great autoscaling solution when you need to scale based on work items held in external systems, like message queues that your application consumes. To some degree, this pattern shares some of the characteristics of Chapter 7, "Batch Job": the workload runs only when work is done and does not consume any resources when idle. Both can be scaled up for parallel processing of the work items. The difference here is that a KEDA ScaledObject does the up-scale automatically, whereas for a Kubernetes `Job`, you must manually determine the parallelism parameters. With KEDA, you can also automatically trigger Kubernetes Jobs based on the availability of external workloads. The ScaledJob custom resource is precisely for this purpose so that instead of scaling up replicas from 0 to 1, a Job resource is started in case a scaler's activation threshold is met. Note that the `parallelism` field in the Job is still fixed, but the autoscaling happens on the Job resource level itself (i.e., Job resources themselves play the role of replicas).

Push Versus Pull Horizontal Autoscalers

Kubernetes knows about two main types of horizontal autoscalers: push autoscalers and pull autoscalers.

Push autoscalers operate by actively pushing metrics to the autoscaler, which then uses those metrics to decide how to scale. This technique is often used when the metrics have been directly generated by a system closely integrated with the autoscaler. For example, in Knative, the Activator pushes the metrics about concurrent requests to the Autoscaler component, as illustrated in Figure 29-2.

Pull autoscalers operate by actively pulling metrics from the application or external sources. Pulling is often used when the metrics are not directly accessible to the autoscaler or when the metrics are stored in an external system. KEDA, for example, is a pull autoscaler that scales deployments based on, for example, the number of events or messages in a queue. Figure 29-3 shows how KEDA uses a custom Kubernetes controller to pull metrics about the number of events and then uses those metrics to determine whether to scale up or down.

Push autoscalers are often used for applications that receive data, like from HTTP endpoints. In contrast, pull autoscalers are suitable for applications that actively retrieve their workload, such as pulling from a message queue.

Table 29-2 summarizes the unique features and differences between HPA, Knative, and KEDA.

Table 29-2. Horizontal autoscaling on Kubernetes

	HPA	Knative	KEDA
Scale metrics	Resource usage	HTTP requests	External metrics like message queue backlog
Scale-to-zero	No	Yes	Yes
Type	Pull	Push	Pull
Typical use cases	Stable traffic web applications, Batch processing	Serverless applications with rapid scaling, serverless functions	Message-driven microservices

Now that we have seen all the possibilities for scaling horizontally with HPA, Knative, and KEDA, let's look at a completely different kind of scaling that does not alter the number of parallel-running replicas but lets your application grow and shrink.

Vertical Pod Autoscaling

Horizontal scaling is preferred over vertical scaling because it is less disruptive, especially for stateless services. That is not the case for stateful services, where vertical scaling may be preferred. Other scenarios where vertical scaling is useful include tuning the resource needs of a service based on actual load patterns. We've discussed why identifying the correct number of Pod replicas might be difficult and even impossible when the load changes over time. Vertical scaling also has these kinds of challenges in identifying the correct `requests` and `limits` for a container. The Kubernetes Vertical Pod Autoscaler (VPA) aims to address these challenges by automating the process of adjusting and allocating resources based on real-world usage feedback.

As we saw in Chapter 2, "Predictable Demands", every container in a Pod can specify its CPU and memory `requests`, which influences where the Pods will be scheduled. In a sense, the resource `requests` and `limits` of a Pod form a contract between the Pod and the scheduler, which causes a certain amount of resources to be guaranteed or prevents the Pod from being scheduled. Setting the memory `requests` too low can cause nodes to be more tightly packed, which in turn can lead to out-of-memory errors or workload eviction due to memory pressure. If the CPU `limits` are too low, CPU starvation and underperforming workloads can occur. On the other hand, specifying resource `requests` that are too high allocates unnecessary capacity, leading to wasted resources. It is important to set resource `requests` as accurately as possible since they impact the cluster utilization and the effectiveness of horizontal scaling. Let's see how VPA helps address this.

On a cluster with VPA and the metrics server installed, we can use a VPA definition to demonstrate vertical autoscaling of Pods, as in Example 29-8.

Example 29-8. VPA

```yaml
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: random-generator-vpa
spec:
  targetRef:               ❶
    apiVersion: apps/v1
    kind: Deployment
    name: random-generator
  updatePolicy:
    updateMode: "Off"      ❷
```

❶ Reference to the higher-level resource that holds the selector to identify the Pods to manage.

❷ The update policy for how VPA will apply changes.

A VPA definition has the following main parts:

Target reference
: The target reference points to a higher-level resource that controls Pods, like a Deployment or a StatefulSet. From this resource, the VPA looks up the label selector for identifying the Pods it should handle. If the reference points to a resource that does not contain such a selector, then it will report an error in the VPA status section.

Update policy
: The update policy controls how the VPA applies changes. The `Initial` mode allows you to assign resource requests only during Pod creation time and not later. The default `Auto` mode allows resource assignment to Pods at creation time, but additionally, it can update Pods during their lifetimes, by evicting and rescheduling the Pod. The value `Off` disables automatic changes to Pods but allows you to suggest resource values. This is a kind of dry run for discovering the right size of a container without applying it directly.

A VPA definition can also have a resource policy that influences how the VPA computes the recommended resources (e.g., by setting per-container lower and upper resource boundaries).

Depending on which `.spec.updatePolicy.updateMode` is configured, the VPA involves different system components. All three VPA components—recommender, admission plugin, and updater—are decoupled and independent and can be replaced with alternative implementations. The module with the intelligence to produce recommendations is the recommender, which is inspired by Google's Borg system. The implementation analyzes the actual resource usage of a container under load for a certain period (by default, eight days), produces a histogram, and chooses a high-percentile value for that period. In addition to metrics, it also considers resource and specifically memory-related Pod events such as evictions and `OutOfMemory` events.

In our example, we chose `.spec.updatePolicy.updateMode` equals `Off`, but there are two other options to choose from, each with a different level of potential disruption on the scaled Pods. Let's see how different values for `updateMode` work, starting from nondisruptive to a more disruptive order:

Off
: The VPA recommender gathers Pod metrics and events and then produces recommendations. The VPA recommendations are always stored in the `status` section of the VPA resource. However, this is as far as the `Off` mode goes. It analyzes and produces recommendations, but it does not apply them to the Pods. This mode is useful for getting insight on the Pod resource consumption without introducing any changes and causing disruption. That decision is left for the user to make if desired.

Initial

In this mode, the VPA goes one step further. In addition to the activities performed by the recommender component, it also activates the VPA admission Controller, which applies the recommendations to newly created Pods only. For example, if a Pod is scaled manually, updated by a Deployment, or evicted and restarted for whatever reason, the Pod's resource request values are updated by the VPA Admission Controller.

This controller is a *mutating admission Webhook* that overrides the `requests` of new matching Pods that are associated with the VPA resource. This mode does not restart a running Pod, but it is still partially disruptive because it changes the resource request of newly created Pods. This in turn can affect where a new Pod is scheduled. What's more, it is possible that after applying the recommended resource requests, the Pod is scheduled to a different node, which can have unexpected consequences. Or worse, the Pod might not be scheduled to any node if there is not enough capacity on the cluster.

Recreate and Auto

In addition to the recommendation creation and its application for newly created Pods, as described previously, in this mode, the VPA also activates its updated component. The `Recreate` update mode forcibly evicts and restarts all Pods in the deployment to apply the VPA's recommendations, while the `Auto` update mode is supposed to support in-place updates of resource limits without restarting Pods in a future version of Kubernetes. As of 2023, `Auto` behaves the same as `Recreate`, so both update modes can be disruptive and may lead to the unexpected scheduling issues that have been described earlier.

Kubernetes is designed to manage immutable containers with immutable Pod `spec` definitions, as seen in Figure 29-4. While this simplifies horizontal scaling, it introduces challenges for vertical scaling, such as requiring Pod deletion and recreation, which can impact scheduling and cause service disruptions. This is true even when the Pod is scaling down and wants to release already-allocated resources with no disruption.

Another concern is the coexistence of VPA and HPA because these autoscalers are not currently aware of each other, which can lead to unwanted behavior. For example, if an HPA is using resource metrics such as CPU and memory, and the VPA is also influencing the same values, you may end up with horizontally scaled Pods that are also vertically scaled (hence double scaling).

We can't go into more details here. Although it is still evolving, it is worth keeping an eye on the VPA as it is a feature that has the potential to significantly improve resource consumption.

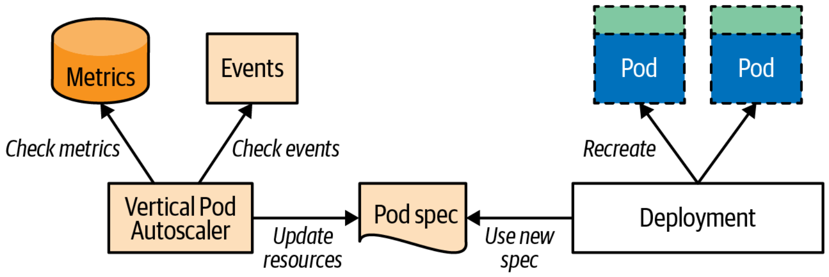

Figure 29-4. Vertical Pod autoscaling mechanism

Cluster Autoscaling

The patterns in this book primarily use Kubernetes primitives and resources targeted at developers using a Kubernetes cluster that's already set up, which is usually an operational task. Since it is a topic related to the elasticity and scaling of workloads, we will briefly cover the Kubernetes Cluster Autoscaler (CA) here.

One of the tenets of cloud computing is pay-as-you-go resource consumption. We can consume cloud services when needed, and only as much as needed. CA can interact with cloud providers where Kubernetes is running and request additional nodes during peak times or shut down idle nodes during other times, reducing infrastructure costs. While the HPA and VPA perform Pod-level scaling and ensure service-capacity elasticity within a cluster, the CA provides node scalability to ensure cluster-capacity elasticity.

Cluster API

All major cloud providers support Kubernetes CA. However, to make this happen, plugins have been written by cloud providers, leading to vendor locking and inconsistent CA support. Luckily, the Cluster API Kubernetes project aims to provide APIs for cluster creation, configuration, and management. All major public and private cloud providers like AWS, IBM Cloud, Azure, GCE, vSphere, and OpenStack support this initiative. This also allows CA to be used in on-premises Kubernetes installations. The heart of the Cluster API is a machine controller running in the background, for which several independent implementations like the Kubermatic machine-controller or the machine-api-operator by Red Hat OpenShift already exist. It is worth keeping an eye on the Cluster API as it may become the backbone for any cluster autoscaling in the future.

CA is a Kubernetes add-on that has to be turned on and configured with a minimum and maximum number of nodes. It can function only when the Kubernetes cluster is running on a cloud-computing infrastructure where nodes can be provisioned and decommissioned on demand and that has support for Kubernetes CA, such as AWS, IBM Cloud Kubernetes Service, Microsoft Azure, or Google Compute Engine.

A CA primarily performs two operations: it add new nodes to a cluster or removes nodes from a cluster. Let's see how these actions are performed:

Adding a new node (scale-up)

If you have an application with a variable load (busy times during the day, weekend, or holiday season and much less load during other times), you need varying capacity to meet these demands. You could buy fixed capacity from a cloud provider to cover the peak times, but paying for it during less busy periods reduces the benefits of cloud computing. This is where CA becomes truly useful.

When a Pod is scaled horizontally or vertically, either manually or through HPA or VPA, the replicas have to be assigned to nodes with enough capacity to satisfy the requested CPU and memory. If no node in the cluster has enough capacity to satisfy all of the Pod's requirements, the Pod is marked as *unschedulable* and remains in the waiting state until such a node is found. CA monitors for such Pods to see whether adding a new node would satisfy the needs of the Pods. If the answer is yes, it resizes the cluster and accommodates the waiting Pods.

CA cannot expand the cluster by a random node—it has to choose a node from the available node groups the cluster is running on.[4] It assumes that all the machines in a node group have the same capacity and the same labels, and that they run the same Pods specified by local manifest files or DaemonSets. This assumption is necessary for CA to estimate how much extra Pod capacity a new node will add to the cluster.

If multiple node groups are satisfying the needs of the waiting Pods, CA can be configured to choose a node group by different strategies called *expanders*. An expander can expand a node group with an additional node by prioritizing least cost or least resource waste, accommodating most Pods, or just randomly. At the end of a successful node selection, a new machine should be provisioned by the cloud provider in a few minutes and registered in the API Server as a new Kubernetes node ready to host the waiting Pods.

Removing a node (scale-down)

Scaling down Pods or nodes without service disruption is always more involved and requires many checks. CA performs scale-down if there is no need to scale up and a node is identified as unneeded. A node is qualified for scale-down if it satisfies the following main conditions:

4 Node groups is not an intrinsic Kubernetes concept (i.e., there is no NodeGroup resource) but is used as an abstraction in the CA and Cluster APIs to describe nodes that share certain characteristics.

- More than half of its capacity is unused—that is, the sum of all requested CPU and the memory of all Pods on the node is less than 50% of the node-allocatable resource capacity.
- All movable Pods on the node (Pods that are not run locally by manifest files or Pods created by DaemonSets) can be placed on other nodes. To prove that, CA performs a scheduling simulation and identifies the future location of every Pod that would be evicted. The final location of the Pods is still determined by the scheduler and can be different, but the simulation ensures there is spare capacity for the Pods.
- There are no other reasons to prevent node deletion, such as a node being excluded from scaling down through annotations.
- There are no Pods that cannot be moved, such as Pods with a PodDisruptionBudget that cannot be satisfied, Pods with local storage, Pods with annotations preventing eviction, Pods created without a controller, or system Pods.

All of these checks are performed to ensure no Pod is deleted that cannot be started on a different node. If all of the preceding conditions are true for a while (the default is 10 minutes), the node qualifies for deletion. The node is deleted by marking it as unschedulable and moving all Pods from it to other nodes.

Figure 29-5 summarizes how the CA interacts with cloud providers and Kubernetes for scaling out cluster nodes.

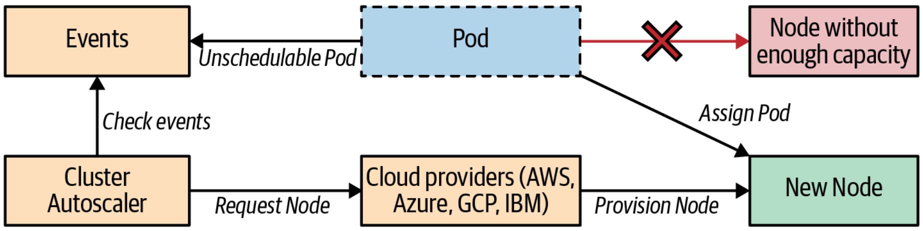

Figure 29-5. Cluster autoscaling mechanism

As you've probably figured out by now, scaling Pods and nodes are decoupled but complementary procedures. An HPA or VPA can analyze usage metrics and events, and scale Pods. If the cluster capacity is insufficient, the CA kicks in and increases the capacity. The CA is also helpful when irregularities occur in the cluster load due to batch Jobs, recurring tasks, continuous integration tests, or other peak tasks that require a temporary increase in the capacity. It can increase and reduce capacity and provide significant savings on cloud infrastructure costs.

Scaling Levels

In this chapter, we explored various techniques for scaling deployed workloads to meet their changing resource needs. While a human operator can manually perform most of the activities listed here, that doesn't align with the cloud native mindset. To enable large-scale distributed system management, automating repetitive activities is a must. The preferred approach is to automate scaling and enable human operators to focus on tasks that a Kubernetes Operator cannot automate yet.

Let's review all of the scaling techniques, from the more granular to the more coarse-grained order, as shown in Figure 29-6.

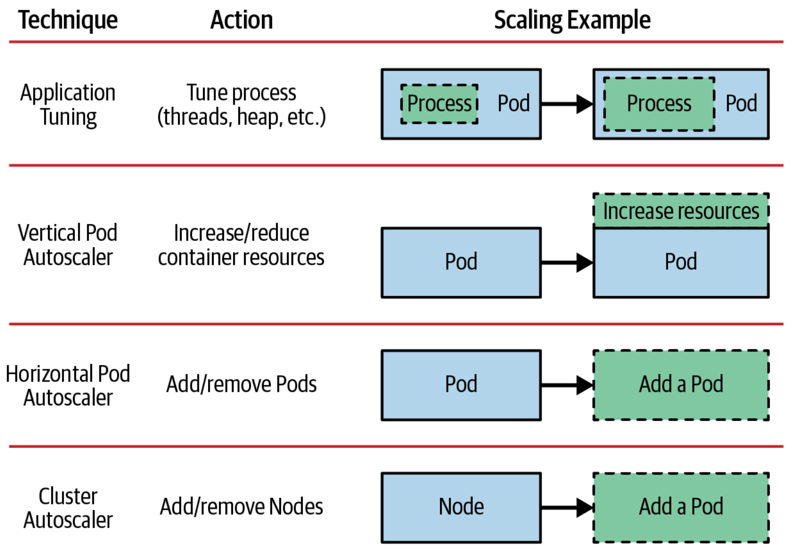

Figure 29-6. Application-scaling levels

Application tuning

At the most granular level, there is an application tuning technique we didn't cover in this chapter, as it is not a Kubernetes-related activity. However, the very first action you can take is to tune the application running in the container to best use the allocated resources. This activity is not performed every time a service is scaled, but it must be performed initially before hitting production. For example, for Java runtimes, that is right-sizing thread pools for best use of the available CPU shares the container is getting, then tuning the different memory regions such as heap, nonheap, and thread stack sizes. Adjusting these values is typically performed through configuration changes rather than code changes.

Container-native applications use start scripts that can calculate good default values for thread counts, and memory sizes for the application based on the allocated container resources rather than the shared full-node capacity. Using such scripts is an excellent first step. You can also go one step further and use techniques and libraries such as the Netflix Adaptive Concurrency Limits library, where the application can dynamically calculate its concurrency limits by self-profiling and adapting. This is a kind of in-app autoscaling that removes the need for manually tuning services.

Tuning applications can cause regressions similar to a code change and must be followed by a degree of testing. For example, changing the heap size of an application can cause it to be killed with an `OutOfMemory` error, and horizontal scaling won't be able to help. On the other hand, scaling Pods vertically or horizontally, or provisioning more nodes, will not be as effective if your application is not consuming the resources allocated for the container properly. So tuning for scale at this level can impact all other scaling methods and can be disruptive, but it must be performed at least once for optimal application behavior.

Vertical Pod autoscaling

Assuming the application is consuming the container resources effectively, the next step is setting the right resource requests and limits in the containers. Earlier, we explored how VPA can automate the process of discovering and applying optimal values driven by real consumption. A significant concern here is that Kubernetes requires Pods to be deleted and created from scratch, which leaves the potential for short or unexpected periods of service disruption. Allocating more resources to a resource-starved container may make the Pod unschedulable and increase the load on other instances even more. Increasing container resources may also require application tuning to best use the increased resources.

Horizontal Pod autoscaling

The preceding two techniques are a form of vertical scaling; we hope to get better performance from existing Pods by tuning them but without changing their count. The following two techniques are a form of horizontal scaling: we don't touch the Pod specification, but we change the Pod and node count. This approach reduces the chances of introducing any regression and disruption and allows more straightforward automation. HPA, Knative, and KEDA are the most popular forms of horizontal scaling. Initially, HPA provided minimal functionality through CPU and memory metrics support only. Now it uses custom and external metrics for more advanced scaling use cases that allow scaling based on metrics that have an improved cost correlation.

Assuming that you have performed the preceding two methods once for identifying good values for the application setup itself and determined the resource consumption of the container, from there on, you can enable HPA and have the application adapt to shifting resource needs.

Cluster autoscaling

The scaling techniques described in HPA and VPA provide elasticity within the boundary of the cluster capacity only. You can apply them only if there is enough room within the Kubernetes cluster. CA introduces flexibility at the cluster capacity level. CA is complementary to the other scaling methods but is also completely decoupled. It doesn't care about the reason for extra capacity demand, or why there is unused capacity, or whether it is a human operator or an autoscaler that is changing the workload profiles. CA can extend the cluster to ensure demanded capacity or shrink it to spare some resources.

Discussion

Elasticity and the different scaling techniques are an area of Kubernetes that is still actively evolving. The VPA, for example, is still experimental. Also, with the popularization of the serverless programming model, scaling to zero and quick scaling have become a priority. Knative and KEDA are Kubernetes add-ons that exactly address this need to provide the foundation for scale-to-zero, as we briefly described in "Knative" on page 317 and "KEDA" on page 321. Those projects are progressing quickly and are introducing very exciting new cloud native primitives. We are watching this space closely and recommend you keep an eye on Knative and KEDA too.

Given a desired state specification of a distributed system, Kubernetes can create and maintain it. It also makes it reliable and resilient to failures, by continuously monitoring and self-healing and ensuring its current state matches the desired one. While a resilient and reliable system is good enough for many applications today, Kubernetes goes a step further. A small but properly configured Kubernetes system would not break under a heavy load but instead would scale the Pods and nodes. So in the face of these external stressors, the system would get bigger and stronger rather than weaker and more brittle, giving Kubernetes antifragile capabilities.

More Information

- Elastic Scale Example
- Rightsize Your Pods with Vertical Pod Autoscaling
- Kubernetes Autoscaling 101
- Horizontal Pod Autoscaling
- HPA Algorithm Details
- Horizontal Pod Autoscaler Walk-Through
- Knative
- Knative Autoscaling

- Knative: Serving Your Serverless Services
- KEDA
- Application Autoscaling Made Easy with Kubernetes Event-Driven Autoscaling (KEDA)
- Kubernetes Metrics API and Clients
- Vertical Pod Autoscaling
- Configuring Vertical Pod Autoscaling
- Vertical Pod Autoscaler Proposal
- Vertical Pod Autoscaler GitHub Repo
- Kubernetes VPA: Guide to Kubernetes Autoscaling
- Cluster Autoscaler
- Performance Under Load: Adaptive Concurrency Limits at Netflix
- Cluster Autoscaler FAQ
- Cluster API
- Kubermatic Machine-Controller
- OpenShift Machine API Operator
- Adaptive Concurrency Limits Library (Java)
- Knative Tutorial

CHAPTER 30
Image Builder

Kubernetes is a general-purpose orchestration engine, suitable not only for running applications but also for building container images. The *Image Builder* pattern explains why it makes sense to build the container images within the cluster and what techniques exist today for creating images within Kubernetes.

Problem

All the patterns in this book so far have been about operating applications on Kubernetes. You've learned how to develop and prepare applications to be good cloud native citizens. However, what about *building* the application itself? The classic approach is to build container images outside the cluster, push them to a registry, and refer to them in the Kubernetes Deployment descriptors. However, building within the cluster has several advantages.

If your company policies allow, having only one cluster for everything is advantageous. Building and running applications in one place can considerably reduce maintenance costs. It also simplifies capacity planning and reduces platform resource overhead.

Typically, continuous integration (CI) systems like Jenkins are used to build images. Building with a CI system is a scheduling problem for efficiently finding free computing resources for build jobs. At the heart of Kubernetes is a highly sophisticated scheduler that is a perfect fit for this kind of scheduling challenge.

Once we move to continuous delivery (CD), where we transition from *building* images to *running* containers, if the build happens within the same cluster, both phases share the same infrastructure and ease transition. For example, let's assume that a new security vulnerability is discovered in a base image used for all applications. As soon as your team has fixed this issue, you have to rebuild all the application images