In Example 24-4 we use the API version v2beta2 of the resource to configure the HPA. This version is in active development and is feature-wise a superset of version v1. Version v2 allows for much more criteria than the CPU usage, like memory consumption or application-specific custom metrics. By using `kubectl get hpa.v2beta2.autoscaling -o yaml`, you can easily convert a v1 HPA resource created by `kubectl autoscale` to a v2 resource.

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. 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. 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*.