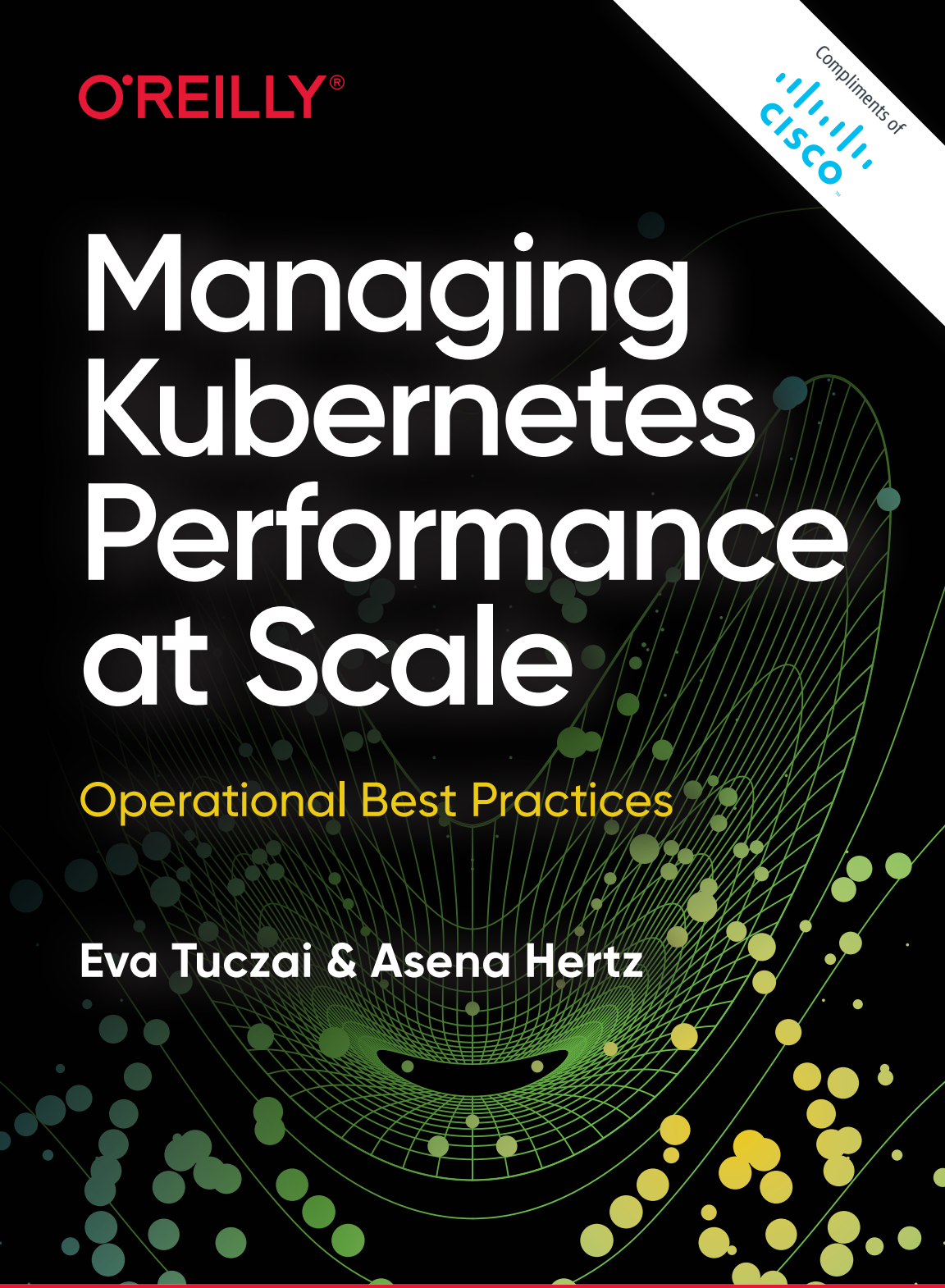
O'REILLY®

Compliments of cisco

Managing Kubernetes Performance at Scale

Operational Best Practices

Eva Tuczai & Asena Hertz



REPORT

Deliver on the promise of Kubernetes

Cisco Workload Optimization Manager adds essential capabilities to Kubernetes deployments so that workloads can be intelligent and self-managed:

- Automate pod rescheduling to ensure performance
- Automatically scale clusters intelligently to implement elastic infrastructure
- Through unified workload automation, control Kubernetes clusters across multicloud environments
- Unite DevOps and infrastructure teams with full-stack visibility

Learn more:

Accelerate cloud-native projects for production-scale Kubernetes with self-managing Kubernetes. Ensure application performance with Cisco Workload Optimization Manager.

Managing Kubernetes Performance at Scale

Operational Best Practices

Eva Tuczai and Asena Hertz

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Managing Kubernetes Performance at Scale
by Eva Tuczai and Asena Hertz

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, please contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Nikki McDonald
Development Editor: Eleanor Bru
Production Editor: Christopher Faucher
Copyeditor: Octal Publishing, LLC
Proofreader: Christina Edwards
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

May 2019: First Edition

Revision History for the First Edition
2019-04-22: First Release
2019-05-29: Second Release
2020-01-23: Third Release

978-1-492-07820-3

[LSI]

Table of Contents

Managing Kubernetes Performance at Scale

Introduction

Enterprises are investing in Kubernetes for the promise of rapid time-to-market, business agility, and elasticity at multicloud scale. Modern containerized applications of loosely coupled services are built, deployed, and iterated upon faster than ever before. The potential for businesses—the ability to bring ideas to market faster—has opened the Kubernetes adoption floodgates. Nevertheless, these modern applications introduce extraordinary complexity that challenges the best of teams. Ensuring that you build your platforms for growth and scale today is critical to accelerating the successful adoption of Kubernetes and the cloud-native practices that enable innovation-first operations.

This ebook is for Kubernetes operators who have a platform-first strategy in their sights, and need to assure that all services perform to meet Service-Level Objectives (SLOs) set by their organization. Kubernetes administrators and systems architects will learn about common challenges and operational mechanisms for running production Kubernetes infrastructure based on proven environments across many organizations. As you learn about the software-defined levers that Kubernetes provides, consider what must be managed by *you* versus what can and should be managed by software.

Building for scale is all about automation. From the mindset and culture to the technologies you adopt and the architectures you introduce, managing elasticity necessitates that IT organizations adopt automation to assure performance without introducing labor

or inefficiency. But automation is not a binary state of you are either doing it or not. Everyone is automating. The crux of automation is the extent to which you allow software to manage the system. From container configuration to autoscaling to full-stack management, there are levers to control things. The question is: are *you* controlling them (deciding what to do and when to do it) or are you letting software do it?

Why Build for Scale Now?

Think about what you're building toward. You want to give developers the agility to quickly deliver business-critical applications and services. You want to assure that the applications *always* perform. And you want to achieve the elasticity required to adapt at scale to continuously fluctuating demands. These are difficult challenges that require the right mindset from the beginning.

Why? Because what you are building can transform the productivity of the lines of business that you support. They will be knocking down your doors to adopt it. In other words, your success accelerates the management challenges that come with greater scale and complexity.

You will not want to say no to new business. Ever. Build and automate for scale now and you won't need to.

Kubernetes Best Practices and the Challenges that Remain

Our targeted audience is someone who uses Kubernetes as a platform for running stateless and stateful workloads in a multitenant cluster, supporting multiple applications or lines of business. These services are running in production, and the operator should take advantage of the data about how these services are running to optimize configuration, dynamically manage allocation of resources to meet SLOs, and effectively scale the cluster capacity in or out to support this demand.

The best practices here focus on how to optimize compute resources for an existing Kubernetes platform and the services running in production. We review how resource allocation in a multitenant environment is managed through quotas and container size specifications, and what techniques are provided within the platform to

manage scaling of resources and services when demand changes. We explore Horizontal Pod, Vertical Pod, and Cluster Autoscaling policies, what factors you need to consider, and the challenges that remain that cannot be solved by threshold-based policies alone.

Still figuring out how you want to build out your Kubernetes platform? Consider reviewing material that discusses how to assure high availability with multiple masters, considerations for the minimum number of worker nodes to get started, networking, storage, and other cluster configuration concepts, which are not covered here.

Managing Multitenancy

Kubernetes allows you to orchestrate and manage the life cycle of containerized services. As adoption grows in your environment, you will be challenged to manage a growing set of services from different applications, each with its own resource demands without allowing workloads to affect one another. Let's first review how containerized services gain access to compute resources of memory and CPU. You can deploy pods without any capacity defined. This allows containers to consume as much memory and CPU that is available on the node, competing with other containers that can grow the same way. Although this might sound like the ultimate definition of freedom, there is nothing inherent to the orchestration of platforms that manages the trade-offs of consumption of resources, against all the workload in the cluster, given the available capacity. Because pods cannot "move" to redistribute workload throughout the cluster, allowing all your services to have untethered access to any resource could cause node starvation, performance issues such as congestion, and would be more complicated to plan for onboarding new services.

Although containers are cattle not pets, the services themselves can be mission critical. You want your cattle to have enough room to graze but not overtake the entire field. To avoid these scenarios, containers can have specifications that define how much compute resources can be reserved for only that container (a request) and the upper capacity allowed (a limit). If you specify both limits and requests, the ratio of these values, whether 1:1 or any:any, changes the Quality of Service (QoS) for that workload. We don't go into detail here about setting limits and requests, and implications such as QoS, but we do explore in the next section the benefits of

optimizing these values by analyzing the actual consumption under production demand.

Even though setting container specifications puts boundaries on our containers, operators will want to manage the total amount of resources allowed for a set of services, to separate what App A can get access to versus App B. Kubernetes allows you to create namespaces (logical groupings in which specific services will run), and you can use other resources for just the deployments in specific namespaces. As the number of services grow, you have an increasing challenge in how to manage the fluctuating demand of all these services and ensure that the pods of one service do not consume a disproportionate amount of resources from the cluster from other services. To manage a multitenant environment and reduce the risk of cluster congestion, DevOps will use a namespace (or project) per team, and then constrain the capacity by assigning resource quotas, which define the maximum amount of resources available to the pods deployed in that namespace. The very use of a resource quota then requires any pod deployed must be minimally configured with a limit or request (whatever matches the resource quota type defined in the namespace). For example, if `myNamespace` has a 10 GiB memory quota limit, all pods running there must have a memory limit specified. You are trading elasticity for control. While these quotas and pod/container specifications provide some guidance on how many resources can be used by a set of workloads, these are now more constraints that have to be monitored, managed, and part of your capacity planning.

Operators can use other techniques to avoid congestion by influencing where pods will be deployed by the scheduler. The use of node labels, affinity and antiaffinity rules, and taints round out the commonly used techniques to apply constraints on the scheduler where pods can run.

In summary, to control how a workload has access to compute resources, the operator can use any one or more of the following techniques to constrain services:

- Namespace quotas to cap limits and requests
- Container specifications to define limits and requests, which also defines QoS
- Node labels to assign workloads to specific nodes

- Affinity/Antiaffinity rules that force compliance of where pods can and cannot run
- Taints and tolerations, as well as considerations for evictions

Container Configurations: Managing Specifications

As previously discussed, you can deploy workloads with specifications that define how much CPU or memory is reserved for a container, defined by requests, and the maximum amount of CPU or memory allowed, as defined by a limit. Before we get into optimizing these values, let's review some use cases in which you want to use limits and requests. Because requests are reserved only for a specific container, you might have services that require a minimum amount of resources to start and run effectively; Java processes come to mind for which you might want to guarantee a minimum amount of memory that can correspond to an `-Xms` (minimum heap) value for the Java process. Likewise, limits present an upper limit that can be intended to prevent a process from using as much memory as it could; for instance, in the case of a memory leak. Limits and requests give you some control over how your workloads are scheduled and run, but you need to set them carefully and adjust them based on how the service actually performs in production.

Next, we explore what you should be doing to manage these constraints in the system.

What happens when a container isn't sized correctly?

Sizing containers appropriately has a direct impact on the end-user experience—and your budget. The implications at scale could make or break the expansion of a Kubernetes platform-first initiative. Let's point out some likely obvious consequences of container sizing.

Although requests provide guaranteed resources for your service, make sure that you are consuming these reservations because these resources are offlimits to any other workload. Being too conservative with requests (allocating too much) has the compound effect over multiple services of requiring more compute nodes to run the desired number of pods, even though the actual overall node consumption is underutilized. The scheduler uses requests as a way to determine capacity on a node; overallocating with requests mainly

assures that you will be overprovisioned, which can also mean you are spending more money.

Additionally, if you are thinking about taking advantage of Horizontal Pod Autoscaling policies, which we discuss in the next chapter, the scheduler can only deploy more workloads onto a node if the node can accommodate all requests of all pods running there. Overallocating request capacity will also guarantee that you must overprovision compute to be able to scale out services.

Let's look at the impact of limits. First, remember that CPU and memory are handled differently; you can throttle CPU, whereas Kubernetes does not support memory swapping. If you have too aggressively constrained the limits, you could starve a pod too soon, or before you get the desired amount of transaction throughput for one instance of that service. And for memory, as soon as you reach 100%, it's OOM (out of memory), and the pod will crash. Kubernetes will assure that a crashed pod will be redeployed, but the user who is waiting for a transaction to complete will not have a good experience leading up to the crash, not to mention the impact the crash has on a stateful service.

Managing vertical scaling of containers is a complicated and time-consuming project of analyzing data from different sources and setting best-guess thresholds. Operators try to mitigate performance risks by allocating more resources just to be safe. Performance is paramount after all. At scale, however, the cost of overprovisioning, especially in the cloud, will delay the successful rollout of your platform-first initiative. You need only look to Infrastructure-as-a-Service adoption for proof: those organizations that struggle with unexpectedly high cloud bills also face delays in adopting cloud-first strategies.

Best practices for sizing containers

When containers are sized correctly, you have assured performance for the transactions running on a containerized service while efficiently limiting the amount of resources the service can access. Getting it right starts with an approximation that needs to be validated through stress testing and production use.

Start your approximations with the following considerations:

1. Is your workload constrained to run in a namespace with a quota? Remember to take your required number of replicas for each service and have the sum fall below your quota, saving room for any horizontal scaling policies to trigger.
2. Do you have a minimum amount of resources to start the service? Define only the minimum. For example, a Java process that has an `-Xms` defined should have a minimum memory request to match that, as long as the `-Xms` value is properly sized.
3. What resource type is your service more sensitive to? For example, if it is more CPU intensive, you might want to focus on a CPU limit, even if it is throttled.
4. How much work is each pod expected to perform? What is the relationship between that work, defined as throughput of requests or response time, and amount of CPU and memory required?
5. Are there QoS guarantees that you need to achieve? You should be familiar with the relationship of limits and requests values and QoS. But don't create divas; burstable QoS will work for mission-critical services. If the service/pod must have a guaranteed QoS, you have to set container specifications so that every memory/CPU limit is equal to the request value, reserving all of the upper limit capacity of resources for that service. Think about that. This may create wasted resources if you are not consuming most of it.
6. You can get some very good resource utilization versus response time data if you create stress-test scenarios to capture this data. Solutions like JMeter and Locust do a great job at defining the test and generating response time and throughput metrics. Container utilization values can come from several sources (cAdvisor and others). One technique is to export these data sources to Prometheus, and then to use something like Grafana to visualize these relationships.

The goal will be to first understand what is a reasonable amount of traffic through one container to ensure that you get a good response time for a minimum number of transactions. Use this data to assess the values you have defined for your containers. You want to specify

enough of a lower limit (requests) to assure the service runs, and then provide enough of an upper limit to service a desired amount of work out of one container Then, as you increase the load, you will be more confident in horizontally scaling this service.

It is very important to reassess whether the container sizing is working for you in production. Use the data that provides insight into real-world fluctuating demand. Ideally, you would want to be able to track every deployment and trend out average, peak consumption of resources against the defined limit and request values. This information is important to determine whether you have oversized containers, or where you continuously reach limits that affect performance or scalability.

Resizing containers has an impact on capacity. Resizing down affords more capacity to other workloads on a node, but it also allows for more pods to be deployed against a namespace quota while using the same cluster resources. Resizing up needs to account for underlying resources, whether the node and namespace has available capacity.

Understanding how to best size containers is important, and requires you to manage the trade-offs of desired performance through one instance of a service, resources available, and fluctuating demand across node and cluster capacity.

Autoscaling

Suppose that you have followed the aforementioned patterns to assure that workloads will not risk other services: set up namespaces with quotas—the requirement placed on any service to specify limits and requests—and you are testing to make sure the container specifications are not too constrained or overallocated. This will help you manage multitenancy, but it does not guarantee service performance when demand increases. What else can you do?

The next direction to look at is how many instances, or replicas, you need to effectively run your service under what you define as a reasonable amount of demand. Like container sizing, gather data on how your services are performing running with a specific number of replicas. Are you getting the correct throughput? Response time? Manually adjust the replica number to see whether you can sustain a predictable and desired SLO. This might require some adjustment

over time, or changing end-user demand, depending on the patterns for your service.

What are the options?

You now have some experience with how your services are performing, but you realize that you still need to accommodate for bursting. You have an idea of a target range of replicas to run your service, but you also want to be able to scale out a bit more should demand warrant it. What is in the Kubernetes arsenal to help take some of the manual effort away? Taking advantage of autoscaling policies.

Autoscaling policies are separated into three categories:

- Horizontally managing services
- Vertically managing container resources
- Node scaling on and off the platform

Horizontal management of services means that you need to express how you want your services to scale in and out based on some pressure, whether resource or SLO based. The goal is to determine the desired number of pods required to run your service when you need them. Today, the mechanism provided within the platform is the Horizontal Pod Autoscaler (HPA) implemented as a Kubernetes API and controller. The functionality is based on defining what metric(s) you want to use, how to get them (aka custom metrics if you want something other than CPU), setting a threshold, and then defining the upper and lower limits of the number of pods you want for a service. We will go into this in more detail in the next section, but it is important to remember that you must configure this trigger-based policy for each service, and that it is based on the average of the service (not per pod). You can find documentation on HPA here.

Vertical management of services means that you want to identify how to vertically scale a container, ideally to manage both requests and limits. Overallocating on a resource affects your ability to scale out and run more services without overprovisioning; underallocating could risk performance. One mechanism that is still in beta is the custom resource definition object called the Vertical Pod Autoscaler (VPA), which manages only resource requests. The algorithm increments request values up and down based on data it is gathering from Prometheus and requires manual configuration to control recommendations. VPA can operate in four modes,

including a recommendation mode (mode = off); assign request values on initial deployment only (initial); and has the ability to change the deployment (recreate versus auto, currently not a significant difference). The VPA project is still in beta (at the time of this ebook) so carefully review the limitations, and consider that there is no correlation between the different policies you create. This could case policy A to undo the benefit of policy B. One example and best practice: you should not use HPA and VPA together for the same service if both policy types are triggered by the same metric. Since VPA can only use CPU or memory, you should consider a custom metric for a HPA policy on the same service. There is a related project called Addon Resizer that requires you to configure how to manage resizing of singletons and heapster, metrics-server, kube-state-metrics addons, and using a "nanny" pod that scales resources. You can find more details on VPA here.

Now let's consider how to manage the nodes or underlying infrastructure. Containerization promises agility, portability, and elasticity. Nodes do not need to be static resources, with the possible exception of baremetal nodes, unless you have the ability to dynamically place blades in and out of service. Proper assessment of demand versus supply of resources can allow you to consider ways to scale the infrastructure. You should consider the following factors:

- Scripts, orchestration to scale nodes, in or out.
- The time it takes to spin up a node.
- When consolidating, assuring capacity is available to handle workload that is being drained.
- Understand whether you have any policies (node labels, taints, etc.) that must be accommodated. The more consistent nodes are, the easier it is to manage resources.
- Monitor memory and CPU, allocatable resources, averages and peaks.

You can manage node resources either on platform using the Cluster Autoscaler (CA) project, which is also part of the Google Kubernetes Engine (GKE Cluster Autoscaler), off platform by using scale groups (autoscaling groups, availability sets, etc.) offered by cloud providers, or setting thresholds tracked from the on-premises infrastructure that needs someone to make a decision about how and where

another worker node can spin up. The main benefit of the CA is that it is watching for a pod pending state that fails due to insufficient resources, which will trigger creating a new node. Likewise, when there is low utilization for an extended period of time and the pods on the lowest utilized node can run elsewhere, a node will suspend. The definition of resources here is also requests. So, as long as you have not overallocated requests, a pod pending state is a good indication of additional compute. But if you have not optimized your container specifications for requests, you could be spinning up additional compute even when, from a consumption perspective, there are resources available in the cluster—this just requires some reorganization. You can find more details on the Cluster Autoscaler here.

If you are using public cloud compute with scale groups, here you will not be able to correlate a pod pending state to the need for more compute. But you could set upper and lower limits of the threshold based on a metric that you might need to configure to be collected, and this threshold would be triggered off of utilization of the resource you specified. This kind of policy can catch when a node is becoming overutilized, or whatever your definition is of that state, but does not guarantee pods will deploy. For more details on scale groups, start here for AWS autoscale groups, Azure availability sets, and Azure scale sets.

Approaches for creating and managing autoscaling policies

HPA policies are the most popular because by definition they do not require a restart of a service, pod, or deployment, unless there is a shutdown of a pod due to underutilization, making these policies more flexible. The desirable outcome of using HPA is a consistent performant service without overprovisioning the environment, which is a difficult task to do. You need to consider and define multiple factors and then test combinations, balancing out the thresholds versus the upper and lower limits of the pods to avoid throwing the environment into a yo-yo pattern of too much (requiring you to overprovision) or too little (which does not assure performance). The process involves asking yourself a series of questions:

1. For what pressure condition am I trying to build an HPA policy?
    a. What are the conditions that most affect the performance of my service? Is it more CPU or memory sensitive?
    b. Do I have SLOs for my service that I want to meet?
    c. If both resources and SLOs are important, will I need to consider how to balance thresholds for multiple metrics in a single policy?
2. What metrics and key performance indicators (KPIs) should I use to best represent this condition?
    a. You can start with one resource KPI, but you will probably realize that you need to also represent a SLO metric of either transaction/request throughput (how many requests can I handle), or response time, or both!
    b. Remember, only the average of the KPI is considered, not the maximum.
    c. SLOs are custom based, so you need to consider how to collect these metrics.
3. What should the KPI threshold be?
    a. Start with a conservative metric that might trigger actions "early" and then iterate to higher thresholds.
    b. When working with multiple metrics, start with each metric separately and then combine.
    c. Remember these values are assessed on the average of a service, so at any given point there will be pods that have higher and lower values.
4. What should I define as the upper and lower limit of number of pods for this service?
    a. Ask yourself whether your goal is to provide reliable, consistent SLO, or to make sure that you have a safety net in case of a burst? Too high of a value can cause a yo-yo of scale up then down then up again.
    b. Does your service have a long startup time? You are likely using readiness and/or liveliness probes, but for services that have a longer "warmup" period, you would want to maintain a higher minimum value to ensure availability.

For more on the topic of how to choose metrics refer to the *Requests-Errors-Duration* pattern (or the RED Method), and Google's Site Reliability Engineering book talks about the *Four Golden Signals*, which are essentially requests, errors, duration, and saturation (utilization).

After you are armed with some data and targeted goals for your service, the process of turning information into an actionable policy is really a cycle of answering questions like what is my SLO, what are my KPIs, and what threshold should I set, and then what are my replica minimum and maximum targets. Figure 1-1 shows the iterative process you need to go through to achieve a scale policy that produces consistent and reproducible results.

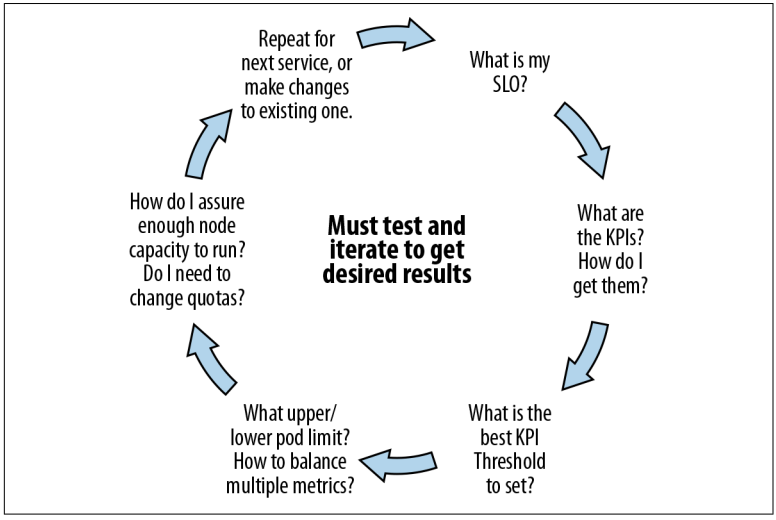


Figure 1. Getting HPA policies right is a continuous exercise that involves time and people.

After you have the combination that balances the outcome that you want to achieve, you need to repeat this process for the next service, and the next. For your first application, and for services that are very similar, the scale of this exercise can be manageable, but as more services want to use HPA, and as services can change in how they behave through different releases, this is a task to which you need to allocate time and people.

As the number of services that utilize HPA policies grow, there are a couple more questions that you must answer: how can I assure that

the infrastructure can support the additional pods being generated, and how can I effectively and dynamically scale my worker nodes?

To answer these questions, you need to consider Node Autoscaling. Start with a simulation that has the maximum number of pods, as defined by the upper limit of your HPA policies, and then assure that this number produces a consistent desired SLO. If it does, you know your upper limit. Don't forget that you have other services running. Simulate your services reaching their upper limits at once. If you end up with a pod pending due to insufficient resources, and without a dynamic way to reclaim unused guarantees (requests) while defragmenting resources, you set a condition to trigger nodes when pods are pending. This is a key use case for the CA Special Interest Group (SIG). This approach gets you out of a bursting situation when you have a flexible infrastructure and can quickly spin up compute. But you need to observe for scenarios where too many nodes spin up and adjust accordingly.

Thresholds do not guarantee performance

The main limitation in working with any threshold-based approach is policies are disparate control points that are not correlated with one another or even able to assess whether an action is executable. A human being needs to consider the possibility that a triggered policy cannot execute, which might then require you to create another trigger. These are myopic scenarios that evaluate only the threshold condition, nothing else. As a backup, you might still need alerting of scenarios requiring human intervention. For example, an action to horizontally scale a service might be triggered off of a policy that has a threshold on response time and CPU, but the new pods might not be assured and can be pending. You then need to be notified that a pod is pending and determine whether the issue is unavailable resources or a namespace quota that was too restrictive to account for scaling. You could implement the Cluster API, which will allow you to set a trigger event for a pod pending to create another worker node, but this would not guarantee that a pod can be scheduled. You might be bound by another constraint of a namespace resource quota or need to make sure that the new node is compliant with everything the pod needs (GPU, labels, Windows or Linux, etc.).

Consider also the methodology of using both HPA and VPA policies, assuming that you avoid using the same threshold metrics for both policy types. At what point do you first want to horizontally

scale a service but risk propagating a not-so-optimized size? Too constrained, and you will just keep running up against the HPA threshold until you reach your pod upper limit count, and performance is still bad. You could end up triggering an HPA policy based on CPU, but then memory requests are oversized, which potentially crowds out other services that need that memory to handle more throughput. You could try to vertically scale first, but how do you know that you have enough node capacity to execute the action?

Although scaling policies are a way to help you provide some level of threshold control coverage to trigger creating more instances of a service, these policies do not test themselves to ensure that they spin up only the right number of pods that can run in the cluster. This means that the operator must consider the "what-if" scenario if the `maxReplicas` defined for the service trigger and other services are also horizontally scaling out at the same time. How do you avoid pending pods? The answer is to set up another scenario to trigger for cluster scaling either based on pod pending (Cluster API SIG addresses this) or based on some node utilization threshold. Does one threshold being triggered from another assure performance? Ideally, you should let the appropriate analysis of the environment work for you: analytics that can assess continuously whether you can resize down to reclaim unused resources, intelligently redistribute running workload, and predict how many pods you would need of each service to maintain response times. You then could even predict the additional nodes that would be needed. The key would be for these decisions to be based on an understanding of the full stack of changing consumption of resources, constraints, and relationships of supply and demand.

Managing the Full Stack

Containerization and microservices provide a benefit to the application developer that they can innovate without concern for the underlying infrastructure. But these containers need to run on some infrastructure that someone somewhere is managing, whether it is on-premises virtualization, public cloud resources, or baremetal nodes. Even Kubernetes is not a fully managed solution: the service provides the convenience of creating the cluster and managing updates, but you still need to manage the compute and storage for which you are paying.

It's important to have insight through all the sources of compute, network, and storage. Bottlenecks below can translate to performance issues above. Think about persistent volumes (PVs) and the associated data store/volume. Knowing that there is input/output per second congestion would be a consideration for the services using that PV.

So how do people get full-stack visibility today? You become comfortable with different tools, and for larger environments, you engage your subject matter expert (SME) peers with more infrastructure background. Even working with SMEs, someone needs to piece together the relationships of the architecture from platform to pods to services, and understand how making a change at one level affects the others. There are a whole host of tools and views from the different layers of the stack: the kubernetes command line interface kubectl, native dashboards (whether K8s or from a PaaS version), if running hosted K8s the public cloud provider views (AKS, EKS, GKE, etc.) are available, the infrastructure views (whether public cloud dashboards or on-premises vSphere client), other related infrastructure like hyperconverged (e.g., Cisco HyperFlex Connect), and more. These SME insights mean that teams have to spend significant amounts of time trying to figure out the dependencies and then determine whether an issue in one layer is being caused by or affecting another. Even overprovisioning, although a costly answer, does not mitigate the need for full-stack insight.

Now imagine that you are multicluster and need a federated view of the environment? How would this operationally scale for multicloud if you want to use different infrastructure and services? Wouldn't you rather be focusing on what is needed to scale the business instead of the number of perspectives of data?

Visibility is not the only objective. Full-stack insight should show you not only components of the platform, but also the relationships and interdependencies. Assuring performance is a complex problem. Even if your answer is to scale out resources, if you are on-premises, placement and capacity of the hypervisor, host, storage, fabric, and so on are factors. Are you in the public cloud? You will exchange these decisions with that of budget and cost.

Compliance

Managing compliance and constraints is actually a full-stack challenge, too. At the top, you might have service-level goals for your applications. In the next layer, you might use techniques to influence placement of workloads that could be for technical or business reasons. Node labels are explicit rules that pods with label x must run on nodes labeled x, as well. These are techniques to guide pods to nodes that provide specific compute capabilities (like GPU processing), or location if there are data sovereignty requirements. This technique is even used to manage licensing and chargeback. Taints and tolerations are more subtle and can be implicit or explicit. You could imply a preference for special pods to be on a node, but if there is pressure in the environment, to loosen that preference. Affinity and antiaffinity rules are more explicit as to what can run where and what can not run. You could also introduce a hard constraint as a compliance rule, such as prescribing the maximum number of pods that can run on a node. Although this is a technique that might (or might not) keep the node from becoming overutilized, it definitely reduces the efficiency of the cluster.

Then, in the layer below, there can be more compliance rules. There can be affinity and antiaffinity rules where compute nodes can run. High availability policies might enforce separation to assure availability in case of a loss.

Decisions made on how to manage resources need insight throughout the stack to assure compliance up and down.

Capacity management

Congratulations! You've rolled out your first set of services using Kubernetes, and you even utilized some of the techniques to influence pod placement, scaling, and stay within business compliance. Don't get too comfortable. The success of this Phase 1 project has opened the floodgates, and now more services want to get onboard. Many more. What's the golden rule? Never keep an application waiting. Even though the pods can deploy in a minute or less, planning for growth can take longer—much longer.

Now you are ready to plan for growth. Borrowing directly from the concepts laid out in this three-part series, "How Full is My Cluster," the first step is to take inventory in how you are managing multitenancy, whether you are using quotas, and then account for the

requests and limits. Requests are important because this is what the scheduler will use to consider available capacity. Because these containers should be treated like cattle, not pets, the variation in the environment is ever changing, so you need some data to understand daily averages and peaks.

Now you collect data from the environment and analyze trends to run scenarios of over- and underestimations. If you have a lot of compliance scenarios to deal with, you might need to assess this on a per-node basis. After you have an estimation that you can live with, you need to assess the additional nodes and resources required against the infrastructure you are managing: on-premises; what host, datastore capacity is required for the additional nodes; if bare metal, where will these blades go, and do you have what is required to accommodate them (maybe lead time is an issue). And in the public cloud, you should be able to articulate the increase in budget needed to run this plan.

This in-depth analysis probably took some time, but it did not account for some other considerations, such as could I have optimized the size of my containers? Did I account for the potential triggering of HPA policies (more pods—assume that they all hit maximum at once or not?)? Do I have more headroom than I thought to accommodate peak demand and can I loosen up on the constraints? These answers require you to rerun your estimations with approximations, which takes more time. And those apps and lines of business are waiting.

A good capacity management strategy should share the same analytics used to manage performance and efficiency in the running environment, which accounts for the full-stack relationships and runs the environment to ensure SLOs are met that optimize workloads' access to the right amount of supply for demand.

Conclusion

Kubernetes promises rapid time-to-market, business agility, and elasticity at multicloud scale. Demand for platforms that allow your lines of business to bring ideas to market faster will quickly grow. What you build today and the best practices you establish will last for years to come. How will you continuously assure performance and maintain compliance while minimizing cost?

By now you know that trying to do this on your own, manually adjusting Kubernetes' software-defined mechanisms, is both labor intensive and risky. Navigating the performance, compliance, and cost trade-offs is not the goal of Kubernetes. There is a reason that a rapidly growing ecosystem of solutions has grown around the platform.

Effectively managing performance, cost, and compliance is a challenge that existed long before Kubernetes and will continue to exist long after. Solving for these trade-offs is critical to your organization's ability to fully achieve the promise of Kubernetes.

This ebook has outlined the software-defined mechanisms that Kubernetes provides. But, again, consider that software should manage these levers, not you. When software continuously and automatically navigates the resource trade-offs that exist in any multicloud environment, you will more quickly reap the benefits of a platform-first strategy. Only software can continuously assure the SLO of each modern application service, elastically adjusting resources as needed while staying compliant. Only you can understand the business and how to best drive it forward.

References

Horizontal Pod Autoscaling

Cluster API (subproject of sig-cluster-lifecycle)

Vertical Pod Autoscaling

Cluster Autoscaler

About the Authors

Eva Tuczai has more than 15 years of experience in IT solutions, including application performance management, virtualization optimization, and automation and cloud native platform integration. As part of Turbonomic's Advanced Engineering team, she is committed to bringing a customer-centric solution approach to solve challenges with performance and efficiency, while leveraging elasticity.

Asena Hertz brings more than a decade of experience in disruptive technologies, spanning workload automation, energy and resource analytics, developer tools, and more. As a Product Marketing leader at Turbonomic, Asena is passionate about the long-term impact and role that cloud native architectures and distributed systems will have on the future of IT and the way businesses bring new ideas to market.