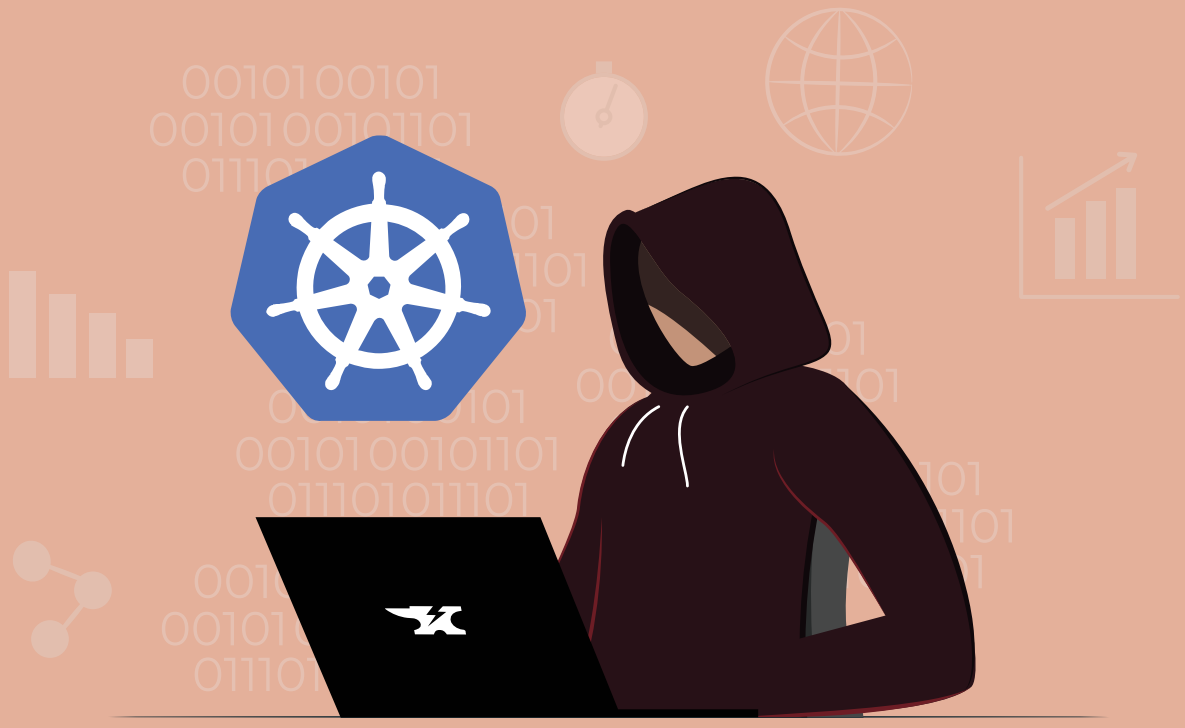
StormForge

Getting Started with Kubernetes Resource Management and Optimization



What's Inside

Introduction

Kubernetes is designed for flexibility, providing fine-grained control for automating deployment, scaling, and management of containerized applications. However, the same flexibility that makes Kubernetes so powerful also introduces complexity. Adopting Kubernetes requires that teams invest time to learn new skills and establish new workflows.

Nowhere are these challenges more apparent than in the world of Kubernetes resource management. The assignment of resources is done at the container level, which pushes capacity decisions to the engineers who are in the best position to understand resource needs. Theoretically, this should drive new levels of efficiency, yet **studies show** that nearly half of cloud resources provisioned (and paid for) go unused.

Let that sink in for a moment. If your company is paying a million dollars a *month* on cloud computing resources, then it's likely that you are wasting **six million dollars** *a year* on resources that are not being used.

So, where's the disconnect? The fact is, setting Kubernetes resources at the right level is not an intuitive or straightforward task. Users are forced to make a choice:

- Estimate conservatively by allocating more resources than expected to allow for uncertainty and variation in actual usage. This results in significant over-provisioning and waste.
- Estimate aggressively and allocate the bare minimum resources in an attempt to minimize cloud costs. This creates the risk of business-impacting performance and reliability issues.
- Spend significant time and effort tuning applications using a manual, trial-and-error approach. This takes valuable engineering resources away from more valuable, business-impacting work, and is ultimately ineffective due to the complexity of the task.

None of the choices is a good one. Fortunately, recent advances in machine learning give us a better approach to finding the best resource settings – both prior to deployment and as things change in production.

We'll cover the ML-based approach in Chapter 3 of this guide, but first we will cover the basics of Kubernetes resources (Chapter 1) and requests and limits (Chapter 2).

After reading this eBook, you should have a solid understanding of the fundamentals of Kubernetes resource management and optimization, including:

- The most common Kubernetes resources, and how they can complicate the ability to optimize applications.
- Kubernetes resource requests and limits, and how they impact quality of service for pods.
- Machine learning techniques that can be used in both non-prod and prod to effectively configure applications for optimal performance and cost-efficiency.

CHAPTER 1:

Understanding Kubernetes Resource Types

Before we dive into Kubernetes resources, let's clarify what the term "resource" refers to here. Anything we create in a Kubernetes cluster is considered a resource: deployments, pods, services and more. For this tutorial, we'll focus on primary resources like CPU and memory, along with other resource types like ephemeral storage and extended resources.

One aspect of cluster management is to assign these resources automatically to containers running in pods so that, ideally, each container has the resources it needs, but no more.

In this article, we'll highlight logical resources for containers running on a cluster. We'll break down four common Kubernetes resources developers work with on a daily basis: CPU, memory, ephemeral storage and extended resources. For each resource, we'll explore how it's measured within Kubernetes, review how to monitor each particular resource and highlight some best practices for optimizing resource use.

Let's explore each primary Kubernetes resource type in depth. Then let's see these resource types in action with some code samples.

CPU

A Kubernetes cluster typically runs on multiple machines, each with multiple CPU cores. They sum up to a total number of available cores, like four machines times four cores for a total of 16.

We don't need to work with whole numbers of cores. We can specify any fraction of a CPU core in 1/1,000th increments (for example, half a core or 500 mill-CPU).

Kubernetes containers run on the Linux kernel, which allows specifying cgroups to limit resources. The Linux scheduler compares the CPU time used (defined by internal time slices) with the defined limit to decide whether to run a container in the next time slice. We can query CPU resources with the `kubectl top` command, invoking it for a pod or node.

We can optimize our use of processor time by making the program running in a container more efficient, either through improved algorithms and coding or by compiler optimization. The cluster user doesn't have much influence on the speed or efficiency of pre-compiled containers.

Memory

The machines in a Kubernetes cluster also each have memory, which again sums up to a cluster total. For example, four machines times 32 **GiB** is 128 GiB.

The kernel level controls main memory, similarly to CPU time with cgroups. If a routine in a container requests memory allocation beyond a hard limit, it signals an out-of-memory error.

Optimizing resource use is largely up to the application's development effort. One step is to improve garbage collection frequency to keep a heap-based image from allocating memory beyond a hard limit. Again, the `kubectl top` command can provide information about memory use.

Exploring CPU and Memory

As our first in-depth example, let's deploy three replicated containers of the popular web server NGINX to a local Kubernetes installation. We're running a one-node "cluster" on our laptop, which only has two cores and 2 GiB of memory.

The code below defines such a pod deployment and grants each of three NGINX containers one-tenth of a core (100 milli-CPU) and 100 MiB of main memory. The code below also limits their use to double the requested values.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          requests:
            cpu: “100m”
            memory: “100Mi”
          limits:
            cpu: “200m”
            memory: “200Mi”
        ports:
        - containerPort: 80
```

We can deploy into the default namespace like this:

```
kubectl apply -f nginx.yaml
```

The local cluster only has a single node. Use this command to return detailed information about it:

```
kubectl describe nodes docker-desktop
```

After clipping most of the output, we can examine some information about resource use:

```
[...]
Namespace                   Name                                CPU Requests
CPU Limits  Memory Requests    Memory Limits  Age
  ---------                   ----                                ------------
----------  ---------------  -------------  ---
  default                    nginx-deployment-585bd9cc5f-djql8    100m (5%)
200m (10%)  100Mi (5%)       200Mi (10%)    66s
  default                    nginx-deployment-585bd9cc5f-gz98r    100m (5%)
200m (10%)  100Mi (5%)       200Mi (10%)    66s
  default                    nginx-deployment-585bd9cc5f-vmdnc    100m (5%)
200m (10%)  100Mi (5%)       200Mi (10%)    66s
[...]
  Resource           Requests     Limits
  --------           --------     ------
  cpu                1150m (57%)  600m (30%)
  memory             540Mi (28%)  940Mi (49%)
  ephemeral-storage  0 (0%)       0 (0%)
  hugepages-1Gi      0 (0%)       0 (0%)
  hugepages-2Mi      0 (0%)       0 (0%)
[...]
```

This information shows the CPU and memory use requests and limits, just as our deployment object specified. It also displays the values as a percentage of the maximum possible allotment.

Next are the current totals for this node, again listed as absolute values and percentages. These numbers include some other containers running in the `kube-system` namespace that we haven't shown here, so there will be a discrepancy not covered by the output above.

The above snippet's last three lines indicate other types of resources beyond CPU and memory, which don't have set requests or limits in this example.

Ephemeral Storage

One additional Kubernetes resource type is ephemeral storage. This is mounted storage that doesn't survive the pod's life cycle. Kubernetes often uses ephemeral storage for caching or logs, but never uses it for important data, like user records. We can request or limit ephemeral storage like main memory, but it's often not as limited a resource.

So what do `hugepages-1Gi` and `hugepages-2Mi` mean in the code snippet above? Huge pages are a modern memory feature of the Linux kernel to allocate large main memory pages of configurable size to processes. We can do this for efficiency.

Kubernetes supports assigning such large pages to containers. These form a resource type per page size that we can request separately.

When specifying a request or limit, we set the total amount of memory, not the number of pages.

```
limits:
  hugepages-2Mi: “100Mi”
  hugepages-1Gi: “2Gi”
```

Here, we limit the number of 2 MiB pages to 50 and the number of 1 GiB pages to 2.

Extended Resources

Cluster users can also define their own resource types — per cluster or node — using the extended resource type. Once we've defined a type and specified available units, we can use requests and limits, just as with the built-in resources we've used so far.

An example is:

```
limits:
  cpu: “200m”
  myproject.com/handles: 100
```

This setting limits the container to 20 percent of a core and 100 of our project's handles.

Resource Requests and Limits

Notice that resource requests and limits were key to our conversation about ephemeral storage and extended resources. This is because an end user can specify resource requests and limits in an application's deployment manifest, which imposes some rules about how Kubernetes should treat a container or pod.

Requests indicate how much of a resource a container should have. They help the scheduler assign pods to nodes based on the amount of resources requested and available resources on those nodes.

Limits are used to indicate a hard upper boundary on how much of a resource a container can use, enforced at the operating-system level. Requests and limits are optional, but if we don't specify a limit, a container can use most of the node's resources, which can have negative cost or performance implications. So, we must be cautious.

Bear in mind that although a pod can contain more than one container, usually there is only one container per pod. We allocate resources to containers, but all of a pod's containers draw from a common pool of resources at the node level.

In the next chapter, we'll dive deeper into the world of Kubernetes requests and limits.

Chapter Summary: Kubernetes Resource Types

Kubernetes clusters maintain hardware resources like CPU time, memory, ephemeral storage and extended resources, and assign them to running containers. Through a system of requests and limits, operators can tailor resource allocation to individual containers, then let the Kubernetes system assign them to nodes appropriately.

Extended resources enable us to define our own resource types and use them similarly. Kubernetes also assigns quality of service designations to pods according to requests and limits. It then uses these designations to make scheduling and termination decisions.

Kubernetes resource optimization is essential to balance costs with the end user experience. Yet, assigning parameters by hand using this article's methods can be time consuming, costly and difficult to scale. We'd rather spend our time creating exciting new features that drive competitive advantage or improved user experience than worrying about optimization and resource use.

In chapter 3, we'll see how machine learning can be used to automate the assigning of parameters, both to free up engineering time but also to make smarter resource decisions. But first, let's dive deeper into the topic of Kubernetes requests and limits.

CHAPTER 2:

Kubernetes Requests and Limits

Kubernetes provides a shared pool of resources that it allocates based on how we configure our containerized applications. The allocation process occurs when a scheduler places pods on nodes. After checking the container's resource configuration, the scheduler selects a node that can guarantee the availability of the resources specified by the container configuration. It then places the container's pod on the suitable node. Typically, this ensures that the deployed microservice will not encounter a resource shortage.

However, some workloads may require more resources than the containerized application has been configured to use. Without a means to restrict the resources that an application can request to use, our application can incur any combination of excessive costs, wasted resources, poor application performance and even application failures. Therefore, it is essential to implement a system of boundaries to prevent resource waste and costly failures.

This is why resource requests and limits are vital components of the Kubernetes environment.

Resource Request

A request refers to the resources that must be reserved for a container. Including a resource request ensures that the container will receive at least what is requested.

When a container includes a resource request, the scheduler can ensure that the node to which it assigns the pod will guarantee the availability of the necessary resources.

Resource Limit

A limit is the maximum amount of resources that a container is allowed to use. When we do not configure a limit, Kubernetes automatically selects one. Therefore, when a limit is poorly configured — whether by a developer or Kubernetes — we can experience any number of unpleasant and costly consequences.

These consequences, which we will explore later in greater depth, include unnecessarily high cloud costs and poor performance due to CPU throttling and out-of-memory (OOM) errors.

While there are several types of resources, in this article we'll focus on CPU and memory. We'll refer to these resources collectively as "compute resources."

CPU and Memory Requests and Limits

CPU

In the Kubernetes environment, CPU represents overall processing power. We measure it in Kubernetes CPUs, where one CPU unit represents one physical CPU or virtual core.

To accommodate a wide variety of application types, we can express CPU measurements with great precision — down to 1/1000th of a CPU or core. If, for instance, we want to request `0.5` of a CPU, we should express it as `500m` — where `m` represents millicores:

```
cpu: “500m”
```

When creating a container, we can use `resources:requests` and `resources:limits` in the manifest to specify CPU resources:

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  namespace: example-namespace
spec:
  containers:
  - name: example-name
    image: repo/example-image
    resources:
      limits:
        cpu: “500m”
      requests:
        cpu: “250m”
```

In the example above, we’ve specified the container’s requested CPU as `250m` and the limit as `500m`. This means the processing power that will be reserved for the container is 250m. Furthermore, if a process requires more than 250m, it can access the additional CPU resources that the scheduler will ensure are available on the node — up to the 500m limit.

MEMORY

Memory resources are measured in bytes and can be expressed as fixed-point numbers, integers, or power-of-two equivalents. We will use the most common type of expression — the power-of-two equivalent, `Mi`. This represents a Mebibyte, or 2^{20} bytes.

For instance:

```
memory: “258Mi”
```

Just as we did with CPU, we can use `resources:requests` and `resources:limits` in the manifest to specify memory resource requests and limits.

Let’s add a memory request and limit to the manifest below:

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  namespace: example-namespace
spec:
  containers:
  - name: example-name
    image: repo/example-image
    resources:
      requests:
        memory: “64Mi”
        cpu: “250m”
      limits:
        memory: “128Mi”
        cpu: “500m”
```

A container created using this manifest will have 64Mi of memory guaranteed and reserved. If a process requires more than that and it is available on the node, it can access the additional amount as long as it doesn’t surpass the 128Mi limit.

Moreover, if a container specifies its compute resource limits but doesn’t specify the resource requests, then Kubernetes will automatically configure the requests to equal the limits.

The Importance of Requests and Limits

Requests are a critical component of pod scheduling, a task executed by Kube-scheduler.

Scheduling is the process of matching pods to the right nodes before the Kubelet runs them. This process accounts for the pod's resource requirements and the free resources on available nodes. In our example — and as is often the case — the pod's resources include the container's required resources and the resources required by the pod itself.

In a cluster, nodes that have the resource capacity to accommodate new pods are called feasible nodes. The scheduler works to evaluate these nodes and select the optimal one for the pod. The assessment process requires the scheduler to consider several factors, including:

- Resource requirements
- Affinity and anti-affinity specifications
- Inter-workload
- Hardware and software constraints

The pod is then placed on the selected node. Then, the Kubelet starts the container inside the pod by passing the container's memory and CPU limits and requests to the container runtime.

If a process inside a container exceeds the requested memory and its node runs out of memory, the pod hosting the process will likely be evicted.

However, it is important to note that there is a sequence to follow when evicting pods. It all depends on the pod's quality of service.

Quality of Service for Pods

Quality-of-Service (QoS) describes a technical system's means of offering different service levels while maintaining the best overall quality, given the hardware's limitations. When a pod is created, Kubernetes assigns a QoS class depending on its resource specifications. There are three classes of QoS:

- **BestEffort:** BestEffort makes no resource availability guarantee. So, it's best suited for applications like batch jobs that can repeat if needed or for staging environments that aren't mission critical. This class is assigned to a pod that has containers without specified memory and CPU requests and limits.
- **Burstable:** The Burstable service level is suitable for pods with a basic use profile that can sometimes increase above the baseline due to increased demand. This level is ideal for databases or web servers, whose load depends on the number of incoming requests. A pod receives this assigned class if one of its containers has a specified memory or CPU request and limit.
- **Guaranteed:** The Guaranteed level offers exactly the requested and limited resources during the pod's lifetime and suits applications like monitoring systems that run at a constant load. A pod is assigned this class if it meets the following conditions:
    - All containers in a pod have memory and CPU requests and limits.
    - All containers have memory requests equal to the limit.
    - All containers have CPU requests equal to the limit.

QoS is important in determining which pods the Kubelet can evict from a node if it runs out of computing resources. An eviction occurs when a node wants to reclaim some resources to ensure the continued operations of the remaining pods.

The Kubelet will evict BestEffort pods before Burstable pods. It will then evict Burstable pods before guaranteed pods.

Consequences of Not Setting Limits and Requests

If we fail to set resource limits and requests, Kubernetes will automatically set them for us. This might sound convenient, but Kubernetes aims to ensure that the application doesn't fail. As a result, it will assign unreasonably generous limits.

For instance, if we do not specify a request or limit for a cluster that only requires 250m of CPU resources, Kubernetes might set a CPU request of 900m. This will bloat the resource requirements of our cluster, making it excessively expensive to run.

Additionally, if Kubernetes unnecessarily reserves large amounts of resources, we might encounter frequent OOM errors.

Furthermore, when the evicted pods accumulate, our clusters become untidy. Even when the evicted pods no longer consume the nodes' resources, they require extra resources from within the Kubernetes environment.

CPU throttling is another consequence of failing to set — or improperly setting — CPU limits. When we specify a CPU limit, Kubernetes attaches a timeframe within which that CPU capacity is allowed. If the container exceeds the limit before the cycle ends, it must pause and wait for the next cycle to begin before it can continue executing. This can lead to significant increases in response time.

Accuracy is paramount in determining requests and limits. Whereas setting the limit too high results in wasted resources, setting it too low dramatically increases the potential for CPU throttling.

Chapter Summary: Kubernetes Requests and Limits

The more resources a Kubernetes cluster consumes, the more expensive it is to run. If we allocate minimal resources to save on cost, we face poor performance and frequent, costly OOM errors.

By properly specifying our compute resource requests and limits, we can minimize overspending, optimize performance, and ensure the most efficient use of our Kubernetes resources. However, trying to determine the optimal balance can be arduous.

In the next chapter, we'll discuss how machine learning-based optimization solutions can be used to find the optimal configuration before deployment, and keep the application performing efficiently after it's in production.

CHAPTER 3:

Using Machine Learning to Automate Kubernetes Optimization

As Kubernetes has become the de facto standard for application container orchestration, it has also raised vital questions about optimization strategies and best practices. One of the reasons organizations adopt Kubernetes is to improve efficiency, even while scaling up and down to accommodate changing workloads. But the same fine-grained control that makes Kubernetes so flexible also makes it challenging to effectively tune and optimize.

In this article, we'll explain how machine learning can be used to automate tuning of these resources and ensure efficient scaling for variable workloads.

The Complexities of Optimization

Optimizing applications for Kubernetes is largely a matter of ensuring that the code uses its underlying resources — namely CPU and memory — as efficiently as possible. That means ensuring performance that meets or exceeds service-level objectives at the lowest possible cost and with minimal effort.

As we've seen, when creating a cluster we can configure the use of two primary resources — memory and CPU — at the container level. Namely, we can set limits as to how much of these resources our application can use and request. We can think of those resource settings as our input variables, and the output in terms of performance, reliability and resource usage (or cost) of running our application. As the number of containers increases, the number of variables also increases, and with that the overall complexity of cluster management and system optimization increases exponentially.

We can think of Kubernetes configuration as an equation with resource settings as our variables and cost, performance and reliability as outcomes.

To further complicate matters, different resource parameters are interdependent. Changing one parameter may have unexpected effects on cluster performance and efficiency. This means that manually determining the precise configurations for optimal performance is an impossible task, unless you have unlimited time and Kubernetes experts.

If we do not set custom values for resources during the container deployment, Kubernetes automatically assigns these values. The challenge here is that Kubernetes is quite generous with its resources to prevent two situations: service failure due to an out-of-memory (OOM) error and unreasonably slow performance due to CPU throttling. However, using the default configurations to create a cloud-based cluster will result in unreasonably high cloud costs without guaranteeing sufficient performance.

This all becomes even more complex when we seek to manage multiple parameters for several clusters. For optimizing an environment's worth of metrics, a machine learning system can be an integral addition.

Machine Learning Optimization Approaches

There are two general approaches to machine learning-based optimization, each of which provides value in a different way. First, **experimentation-based optimization** can be done in a non-prod environment using a variety of scenarios to emulate possible production scenarios. Second, **observation-based optimization** can be performed either in prod or non-prod, by observing actual system behavior. These two approaches are described next.

Experimentation-based Optimization

Optimizing through experimentation is a powerful, science-based approach, because we can try any possible scenario, measure the outcomes, adjust our variables and try again. Since experimentation takes place in a non-prod environment, we're only limited by the scenarios we can imagine and the time and effort needed to perform these experiments. If experimentation is done manually, the time and effort needed can be overwhelming. That's where machine learning and automation come in.

Let's explore how experimentation-based optimization works in practice.

STEP 1: IDENTIFY THE VARIABLES

To set up an experiment, we must first identify which variables (also called parameters) can be tuned. These are typically CPU and memory requests and limits, replicas and application-specific parameters such as JVM heap size and garbage collection settings.

Some ML optimization solutions can scan your cluster to automatically identify configurable parameters. This scanning process also captures the cluster's current, or baseline, values as a starting point for our experiment.

STEP 2: SET OPTIMIZATION GOALS

Next, you must specify your goals. In other words, which metrics are you trying to minimize or maximize? In general, the goal will consist of multiple metrics representing trade-offs, such as performance versus cost. For example, you may want to maximize throughput while minimizing resource costs.

Some optimization solutions will allow you to apply a weighting to each optimization goal, as performance may be more important than cost in some situations and vice versa. Additionally, you may want to specify boundaries for each goal. For instance, you might not want to even consider any scenarios that result in performance below a particular threshold. Providing these guardrails will help to improve the speed and efficiency of the experimentation process.

Here are some considerations for selecting the right metrics for your optimization goals:

- If a containerized application is transaction-based, minimize the response time and the error rate. In this situation, maximum speed is the ideal, and resource use is of less concern.
- If the application is only meant for computations, minimize the error rate. We want to optimize for performance efficiency.
- If the application processes data, speed is likely secondary. Optimize for cost.

Of course, these are just a few examples. Determining the proper metrics to prioritize requires communication between developers and those responsible for business operations. Determine the organization's primary goals. Then examine how the technology can achieve these goals and what it requires to do so. Finally, establish a plan that emphasizes the metrics that best accommodate the balance of cost and function.

STEP 3: ESTABLISH OPTIMIZATION SCENARIOS

With an experimentation-based approach, we need to establish the scenarios to optimize for, and build those scenarios into a load test. This might be a range of expected user traffic or a specific scenario like a retail holiday-based spike in traffic. This performance test will be used during the experimentation process to simulate production load.

STEP 4: RUN THE EXPERIMENT

Once we've set up our experiment with optimization goals and tunable parameters, we can kick off the experiment. An experiment consists of multiple trials, with your optimization solution iterating through the following steps for each trial:

1. The experiment controller runs the containerized application in your cluster, using the baseline parameters for the first trial.
2. The controller then runs the performance test created previously to apply load to the system for our optimization scenario.
3. The controller captures the metrics corresponding to our goals, for example, duration and resource cost.
4. The machine learning algorithm analyzes the results and then calculates a new set of parameters for the next trial.
5. This process is then repeated for however many trials were specified when configuring your experiment. Typical experiments range from 20 to 200 trials, with more parameters requiring more trials to get a definitive result.

The machine learning engine uses the results of each trial to build a model representing the multi-dimensional parameter space. In this space, it can examine the parameters in relation to one another. With each iteration, the ML engine moves closer to identifying the configurations that optimize the goal metrics.

STEP 5: ANALYZE RESULTS

While machine learning automatically recommends the configuration that will result in the optimal outcomes, additional analysis can be done once the experiment completes. For example, you can visualize the trade-offs between two different goals, see which parameters have a significant impact on outcomes and which matter less.

Results are often surprising, and can lead to key architectural improvements, for example determining that a larger number of smaller replicas is more efficient than a smaller number of "heavier" replicas.

Experiment results can be visualized and analyzed to fully understand system behavior.

Observation-based Optimization

While experimentation-based optimization is powerful for analyzing a wide range of scenarios, it's impossible to anticipate every possible situation. Additionally, highly variable user traffic means that an optimal configuration at one point in time may not be optimal as things change. **Kubernetes autoscalers** can help, but they are based on historical usage and fail to take application performance into account.

This is where observation-based optimization can help. Let's see how it works.

STEP 1: CONFIGURE THE APPLICATION

Depending on what optimization solution you're using, configuring an application for observation-based optimization may consist of the following steps:

- Specify the namespace and, optionally, a label selector, to identify which resources to tune.
- Specify guardrails (min and max) for the CPU and memory parameters to be tuned.
- Specify how frequently the system should recommend updated parameter settings.
- Specify whether to deploy recommendations automatically or with approval.

STEP 2: MACHINE LEARNING ANALYSIS

Once configured, the machine learning engine begins analyzing observability data collected from **Prometheus**, **Datadog** or other observability tools to understand actual resource usage and application performance trends. The system then begins making recommendations at the interval specified during configuration.

STEP 3: DEPLOY RECOMMENDATIONS

If you specified automatic implementation of recommendations during configuration, the optimization solution will automatically patch deployments with recommended configurations as they are recommended. If you selected manual deployment, you can view the recommendation including container-level details before deciding to approve or not.

Optimization Best Practices

As you may have noted, observation-based optimization is simpler than experimentation-based approaches. It provides value faster with less effort, but on the other hand, experimentation- based optimization is more powerful and can provide deep application insights that aren't possible using an observation-based approach.

Which approach to use shouldn't be an either/or decision; both approaches have their place and can work together to close the gap between prod and non-prod. Here are some guidelines to consider:

- Because observation-based optimization is easy to implement and improvements can be seen quickly, it should be deployed broadly across your environment.
- For more complex or critical applications that would benefit from a deeper level of analysis, use experimentation-based optimization to supplement observation-based.
- Observation-based optimization can also be used to identify scenarios that warrant the deeper analysis provided by experimentation-based optimization.
- Then use the observation-based approach to continually validate and refine the experimentation-based implementation in a virtuous cycle of optimization in your production environment.

Using both experimentation-based and observation-based approaches creates a virtuous cycle of systematic, continuous optimization.

Chapter Summary: Using Machine Learning to Automate Kubernetes Optimization

Optimizing our Kubernetes environment to maximize efficiency (performance versus cost), scale intelligently and achieve our business goals requires:

1. An ideal configuration of our application and environmental parameters prior to deployment
2. Continuous monitoring and adjustment post-deployment

For small environments, this task is arduous. For an organization running apps on Kubernetes at scale, it is likely already beyond the scope of manual labor. Fortunately, machine learning can bridge the automation gap and provide powerful insights for optimizing a Kubernetes environment at every level. A complete approach to ML-based optimization uses both experimentation in non-prod environments and observation in prod.

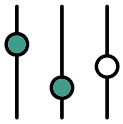
StormForge: A Complete Solution For Kubernetes Resource Optimization

StormForge uses patent-pending machine learning to optimize Kubernetes environments, reducing cloud costs, improving application performance, and providing deep application insights to drive architecture improvements. StormForge automates Kubernetes resource efficiency at scale, accelerating your competitive advantage by allowing developers to focus on innovating, not tuning or troubleshooting Kubernetes.

The StormForge platform includes three solutions that together close the loop between prod and non-prod, ensuring performance and cost-efficiency.

StormForge Optimize Live: Turn observability into actionability

- Observation-based optimization in prod or non-prod
- Use with all applications for fast and easy efficiency gains
- Leverages observability data already being collected
- ML continuously recommends CPU & memory to optimize in real-time
- Recommendations can be automatically implemented or manually approved
- Simple configuration, fast time-to-value



StormForge Optimize Pro: Proactive optimization with deep application insights

- Experimentation-based optimization in non-prod
- Use with complex, mission-critical applications
- Load testing used to simulate range of scenarios
- ML optimizes for any goal by tuning any parameter
- Highlights trade-offs to enable smart business decisions
- Provides deep application insights to drive architectural improvements

StormForge Performance Testing: Scalable, easy-to-use Kubernetes load testing

- Kubernetes performance testing as a service
- Use with Optimize Pro for non-prod scenario planning and optimization
- Get started creating load tests in minutes
- Scale to to hundreds of thousands of requests per second, millions of concurrent users
- Built for automation into CI/CD workflow
- Open workload model for accurate, real-world scenarios

The Value of Optimizing Applications with StormForge

Improve resource utilization and reduce cloud costs, in many cases by 50% or more

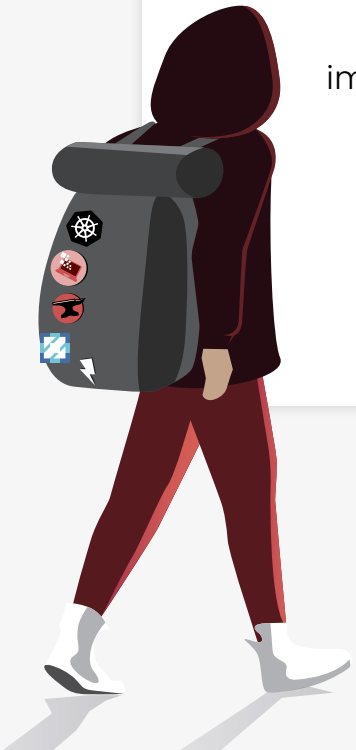
- Reduce risk of application performance and availability issues
- Increase developer productivity by eliminating manual, trial-and-error tuning
- Identify key architectural improvements with in-depth application insights
- Elevate your game and overcome the Kubernetes resource and skills gap

Explore StormForge

Discover how StormForge can help automatically improve application performance and cost efficiency in cloud native environments.

REQUEST A TRIAL →

SCHEDULE DEMO →



+1 (857) 233-9831 | info@stormforge.io | stormforge.io