StormForge

Kubernetes Resource Management at Scale

An ML-Driven Approach to Optimization

This page is intentionally left blank.

Table of Contents

Introduction

Among other strengths, Kubernetes is known for its out-of-the-box load sharing and performance management capabilities. Kubernetes validates the resource requirements of our pods and is generally apt at matching them with appropriately provisioned nodes, monitoring cluster health, and rescheduling pods when nodes are no longer viable.

This dynamic provisioning offers a basic level of scalability for our workloads, which we can't always perform manually upfront, but to scale confidently, we need a more comprehensive approach. The `kube-scheduler`, which is responsible for distributing pods in our clusters, is a useful starting point. However, tuning our clusters' behavior and efficiently managing their resources depends on optimizations beyond the capabilities of a scheduling process.

To guarantee that our Kubernetes clusters and the applications running on them are healthy and effectively optimized, we should integrate active resource management into our day-to-day cluster administration and observability processes. Real-time visibility and insight into resource consumption help avoid CPU throttling, outages caused by memory constraints, and other resource-driven performance issues. And, as we gather more insight into the behaviors and resource usage patterns of our Kubernetes clusters, we'll find that our Kubernetes-backed applications run more effectively and scale more efficiently — leading to a welcome reduction in operating expenses.

This white paper covers the core aspects of Kubernetes resource management. We'll start by explaining how the main Kubernetes resources interact on a node and impact the pods to which they're allocated. We'll follow with some best practices for configuring resources for pods and containers and then examine the implications for scaling and resource management at the cluster level.

Then, we'll discuss the roles of machine learning-based analysis and automated configuration tuning before highlighting the capabilities of StormForge: an intelligent scaling and optimization solution that automates resource management and optimizes Kubernetes workloads and applications. StormForge improves resource efficiency and performance through machine learning, load-testing, and intelligent action, based on the analysis of observability data from our Kubernetes clusters.

Understanding Kubernetes Resource Usage

To understand the factors that contribute to Kubernetes resource consumption, let's begin with a diagram illustrating each of the main components in a Kubernetes architecture:

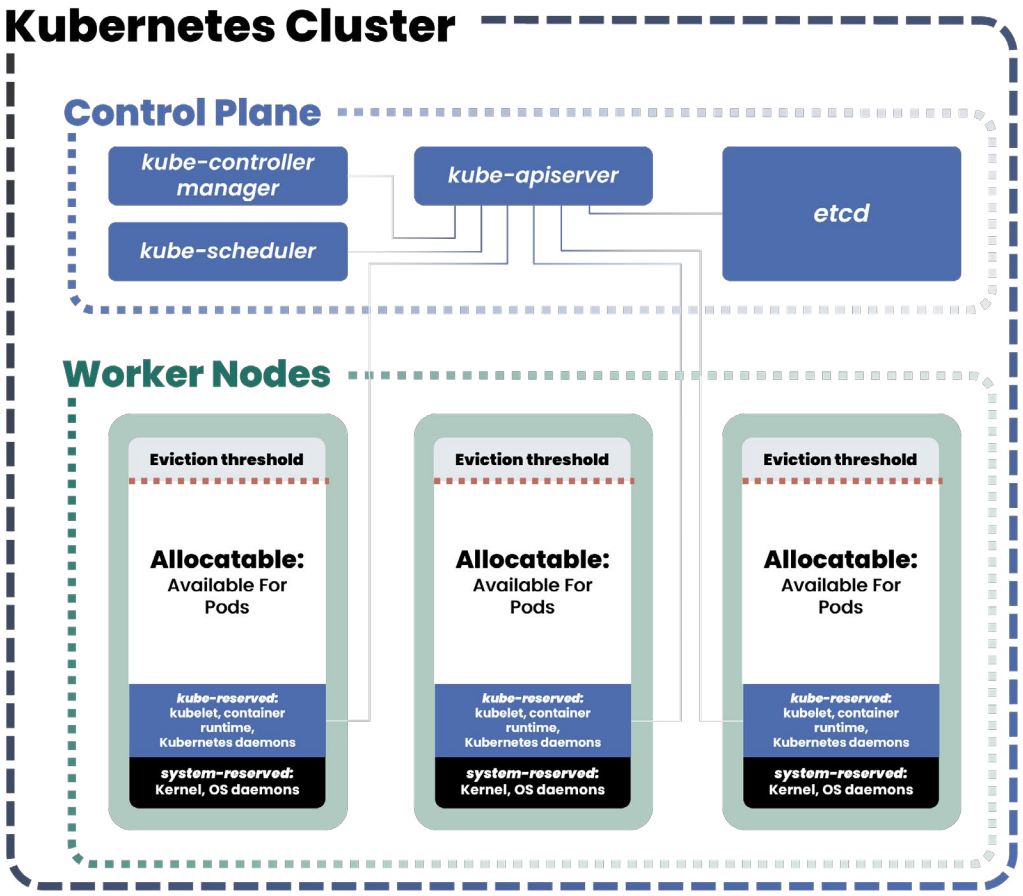

Figure 1. Key Kubernetes Architecture Components

At the top of the architecture are the control plane components that manage each cluster. The control plane typically runs on one or more dedicated machines in the cluster and contains components like the API server, scheduler, and controllers.

The scheduler selects nodes for pods in the scheduling queue one at a time, in a two-step process of filtering and scoring. First, the scheduler finds nodes that are feasible for running the pod by comparing available node resources and characteristics — such as data locality, inter-workload interference, and affinity specifications — against the pod's requests and affinities. It then identifies the most suitable node and instructs the kubelet on that node to interface with the container runtime and start the pod's containers.

A Kubernetes node is a physical or virtual machine that hosts the Kubernetes runtime components. Ideally, each node in the cluster runs on hardware with identical processors, as Kubernetes ignores processor performance when allocating processing resources. In practice, however, Kubernetes is capable of supporting mixed compute resources within the same cluster. For example, we can repurpose older physical servers and integrate them with newer, more powerful servers with different CPU, memory, and storage characteristics. When we mix architectures or operating systems, it's good practice to identify the differences by labeling our nodes so we can use **node affinity** rules to simplify node scheduling.

Kubernetes nodes themselves run several components:

- The OS system daemons and Kubernetes system daemons.
- The kubelet, which manages communications between the control plane and the node(s), directing the container runtime to start or stop containers.
- A container runtime like containerd. Docker was once the most popular runtime, but the Docker Engine is now **deprecated** in favor of runtimes that use the CRI or OCI standards, which support features like cgroups v2 and user namespaces.
- The pods and containers running our application workloads.

Although the eviction thresholds on nodes are not considered a separate component, it's helpful to treat them as such when provisioning for our workloads. We set eviction thresholds to ensure nodes proactively begin freeing up resources when they approach full capacity usage. This helps avoid starving our nodes if they experience surges in resource requests and provides a more controlled, graceful eviction process.

This white paper focuses on the largest section of each node in the diagram: the allocatable portion of its capacity. We'll explore which resources and practices are most critical to maintaining the overall health, efficiency, and cost-effectiveness of our Kubernetes clusters. We'll also contextualize the discussion with robust monitoring and observability practices to help scale our workloads. This includes examining the CPU and memory consumption specifications for a whole node to see how efficiently our deployments run.

Kubernetes Resources

To effectively manage our Kubernetes resources, we must regulate the allocation and usage of the four primary compute resources: CPU, memory, local ephemeral storage, and extended resources. We typically start with high availability as our goal when examining the health of our pods, but should also administer resources with cost, performance and scalability in mind.

First, let's examine a sample configuration file that specifies CPU, memory, and local ephemeral storage. Note the values for `requests` and `limits`.

```
---
apiVersion: v1
kind: Pod
metadata:
  name: webapplication
spec:
  containers:
  - name: samplewebapp
    image: mysamplewebapp.company.img/app:latest
    resources:
      requests:
        memory: "16Mi"
        cpu: "250m"
        ephemeral-storage: "1Gi"
      limits:
        memory: "64Mi"
        cpu: "500m"
        ephemeral-storage: "2Gi"
  - name: database
    image: samplemysql.company.img/dbase:latest
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Requests and Limits

The primary way to reliably provision our clusters is by specifying requests and limits for our containers. Requests specify the minimum compute resources a container requires, and limits determine the maximum resources that a node can allocate to a container.

To meet the basic criteria for binding a pod to a node, the scheduler sums the request values of a pod's containers and compares them against the allocatable resources on the node. Features like pod Quality-of-Service (QoS) classes, which we will discuss in the **Quality of Service** section, rely on these aggregate values. If a pod's request is greater than the available resources on any node in our cluster, the scheduler will hold the pod in the Pending state indefinitely.

A container can consume additional resources beyond its request values, up to the container's specified limits, as long as the node has available resources. This lets us specify a range within which a container can operate with increased performance. Typically, this is meant for bursts, but, in theory, we could run a container that consumes more than its requested memory indefinitely if there are no pods with competing containers on the node.

CPU Resource Units

CPU resources in Kubernetes are measured in CPU units, with each unit representing one physical or virtual CPU core. We can specify CPU allocations in fractional values (either in decimal or millicores). For example, 0.21 CPU and 210m CPU are equivalent — and they are absolute, not relative values.

Containers attempting to allocate CPU time above their limits are throttled. If we specify a CPU limit but no request, Kubernetes assigns a CPU request equal to the container's limit. If we don't specify a limit, the container can potentially use all the CPU resources on the node — with the exception of any cores made exclusively available to a pod with a Guaranteed QoS.

Memory Resource Units

Kubernetes configures memory allocation in bytes, which can be expressed as integers or fixed-point quantities using the suffixes E, P, T, G, M, and k, (for exabyte, petabyte, etc.) or using the equivalent power-of-two suffixes: Ei, Pi, Ti, Gi, Mi, Ki . For example, instead of specifying the memory allocation as 1048576, we can express this value as 1M (for 1 megabyte). The specific allocation and reservation process depend on the node's operating system.

When a container starts, the container runtime configures a kernel cgroup to enforce our configured limits. If a container tries to allocate more memory than is available for that cgroup, the Linux kernel's built-in out-of-memory management process (OOM Killer) terminates the responsible processes, which typically terminates the container. If the container is restartable, the kubelet restarts it.

As with CPU, if we specify a memory limit but no request, Kubernetes automatically assigns a memory request equal to the container's limit. If we don't specify a limit, the container can potentially use all the memory available on the node and trigger an OOM Kill. Containers with no memory limits are more likely to be killed.

Note that OOM conditions are not particularly a concern in Windows environments because there's no way to run privileged containers in Windows (the privileged designation essentially grants host capabilities to the container). Containers always run with a system namespace filter in place, so Windows nodes don't overcommit memory for processes. Instead, Windows treats memory allocations as virtual and relies on disk paging when memory is over-provisioned. Our pods are likely to suffer some performance issues, but processes aren't terminated due to OOM conditions.

Ephemeral Storage

In addition to CPU and memory, Kubernetes nodes also provide running Pods with local ephemeral storage, measured in bytes. Ephemeral storage provides no guarantee of durability for the volume and its data, which are typically lost when the containers using them restart.

Local ephemeral storage is used for transient data like caches, logs, and swap space, and to gracefully stop and restart pods without needing to use **PersistentVolumes**. We can also use ephemeral storage to run ephemeral containers in the event we need to inspect running Pods that can't otherwise accommodate additional containers.

A container with a writable layer and log usage above its assigned limit causes the kubelet to mark its pod for eviction. The kubelet also monitors each pod's `emptyDir` volumes and adds them to the total ephemeral storage usage of the pod's containers to determine when to evict the pod.

If we turn off the **LocalStorageCapacityIsolation** feature, the kubelet doesn't monitor local ephemeral storage allocation and doesn't evict pods — although it does cause the node to taint itself as short on local storage, which may trigger evictions for pods that don't tolerate this condition.

Extended Resources

Cluster operators can also provide fully qualified resource names for resources external to Kubernetes. Some extended resources are device plugin-managed, including GPUs and FPGAs, high-performance networking components, and other devices requiring vendor-specific setup. Other extended resources, such as blockchain USB dongles or software-only resources, can be advertised through the API server. In both cases, extended resources are restricted to whole-number quantities as opaque integer resources.

Quality of Service

Kubernetes assigns each pod one of three Quality of Service (QoS) classes before binding it to a node. The QoS class a pod is given depends on the requests and limits we specify for its containers.

Guaranteed

For a pod to be assigned a Guaranteed QoS class, every container in the pod must have a CPU request equal to its CPU limit and a memory request equal to its memory limit. Because the scheduler assigns nodes based on resource requests, this has the effect of enforcing that a Guaranteed pod will always be created on a node that can allocate enough resources to meet the pod's maximum demands. Recall that if a container specifies a memory or CPU limit, but not a request, Kubernetes automatically assigns values to the missing memory or CPU request that match their corresponding limits.

Containers with integer CPU requests in pods with a Guaranteed QoS class can also be allocated the use of **exclusive CPUs**.

Burstable

Pods that don't meet the criteria for a Guaranteed QoS class label but include at least one container with a memory or CPU request or limit are assigned a Burstable QoS class. The scheduler doesn't guarantee that pods with this classification will be placed on nodes that meet their maximum resource needs, but the node can allocate additional resources to containers requesting them, up to the container's limits.

BestEffort

Pods that have no specified memory or CPU requests or limits are assigned a BestEffort QoS class. Containers in pods with this classification may consume any available resources on the node for as long as they are available but are at risk of not getting the resources they need. However, BestEffort pods can be useful for discovering our pod workload behavior dynamics during ambiguous resource availability conditions.

QOS Implications

QoS classes are primarily descriptive categories of a pod's resource requests. We can use QoS classes to estimate the order in which the kubelet *might* evict pods, but the **main determinants** of pod eviction order, listed here, are not directly related to QoS class:

1. Whether a pod's usage of the resource causing starvation exceeds requests
2. **Pod priority**
3. Resource usage relative to requests

This means that Guaranteed pods and Burstable pods using less than their requested resources will never be evicted in order to make resources available to other pods. The kubelet only evicts these types of pods when accommodating a system daemon that's using more resources than are allocated for it — and when the node only has these types of pods remaining.

QoS class *does* affect the order in which containers are killed under OOM conditions: the OOM Killer first kills containers in pods with lower QoS classes and those that consume memory in excess of their requests.

Consider the implications of these characteristics. Although BestEffort pods present the most obvious resource starvation risk — potentially using a node's entire resources, preventing new pods from being scheduled on the node, or causing OOM conditions — we should be careful to monitor Guaranteed and Burstable pods as well. Resource provisioning can involve a lot of guesswork, and over-provisioned pods reserve valuable unused capacity for as long as they remain on the node. Although the applications and services may be highly available, the inefficiencies introduced by over-provisioning can become costly.

Additionally, Guaranteed pods scheduled to run on nodes under memory pressure can effectively push lower-priority or lower-QoS class pods off the node. This behavior isn't always undesirable, but it can have unexpected consequences.

Our Kubernetes operations teams may need to plan for diverse responses to these situations. In a scenario with static or known scalability, the scheduler's default behaviors may be enough to meet our needs. However, during unpredictable workloads or spikes in usage, we should take into account that resource allocation at the pod level may be insufficient when considering all the other factors influencing scheduling. For example, depending on our application's needs, our cluster may not have enough nodes available to scale our workload reliably. So, it's important to consider the resource demands of our entire cluster(s).

Kubernetes Resource Management Best Practices

Now that we've examined the key concepts to focus on when managing our cluster resources, let's explore some best practices for managing and allocating resources.

Specify Quotas for Resources

In addition to allocating resource limits for individual pods, we can also define resource quotas at the Kubernetes namespace level by using the `ResourceQuota` object. A resource quota defines limits for compute resource usage for a specific namespace, among other characteristics like the number of pods and services that can be deployed.

Use Namespaces

In clusters with more than tens of users, we should use namespaces to isolate groups of resources. This allows us to define each resource's quota settings in `ResourceQuota` on a per-namespace basis. Namespaces should be descriptive and different for each team. Pods and services are always part of a namespace, so a distinctive namespace lets the quota system track behavior and more reliably enforce resource limits.

Configure Allocatable Space

By design, a node's full capacity is available for allocation to pods. Even before we begin considering how to configure pod resource allocation, we should reserve compute resources for a node's system and Kubernetes daemons. Otherwise, they'll compete with pods for resources and starve our nodes — especially as we begin scaling our workloads.

The kubelet lets us manage resources across a node as a whole using the default Node Allocatable feature and the optional `kube-reserved` and `system-reserved` flags. We can specify soft or hard thresholds at which the kubelet tries to reclaim resources. The kubelet first attempts to reclaim node-level resources. If this doesn't bring allocated resources under the threshold, the kubelet **ranks pods** and begins setting their `PodPhase` to `Failed` to terminate them.

Resource Configuration

The Kubernetes resource configuration files themselves also simplify resource management. We can group different resources together in the same configuration file so they can be deployed together.

Let's examine a sample configuration file written in YAML to discuss configuration and labels:

```
apiVersion: v1
kind: Service
metadata:
  name: sample-nginx-service
  labels:
    app: nginx
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-nginx-deploy
  labels:
    app: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.21.6
        ports:
        - containerPort: 80
```

We can configure high availability for our pods and services using the `replicas` parameter in our YAML or Helm charts. This parameter specifies how many copies of a pod we want to have running all the time, flagging the service as highly available. The scheduler then spreads the replicas across different nodes.

For example, we can configure a front-end web app container and a load balancing service in the same deployment file by separating each component with a line containing three dashes (---). Each component gets deployed in the order that they appear in the file — for example, services the pods depend on appear first. This way, the scheduler can properly distribute the pods as the controller creates them. We trigger the deployment with the `kubectl apply` command, as we would to run a single-item deployment.

Label Kubernetes Objects

We can use the built-in label feature to help us track all the objects in our clusters and more accurately determine what resources they're consuming.

Labels are key-value pairs that allow us to add identifying metadata to our Kubernetes objects. In the sample configuration file above, `app: nginx` represents a label. We can use labels to organize these objects or select them from across our Kubernetes environment — for example, by constraining which nodes a pod can be scheduled on. They also let us apply bulk operations to objects.

For example, to run a simple query selecting canary release pods running in the production environment, we first make sure we have applied the labels `environment: production` and `release: canary`. We then use kubectl and run the following query with the `-l` (label) parameter:

```
kubectl get pods -l  environment=production,release=canary
```

We can target some resources — a Job, Deployment, DaemonSet, or ReplicaSet — in configuration files using a selector with more expressive set-based requirements. We can also use a REST client to target any resource. For example, we can use kubectl to target the API server to list the same pods as in the previous command:

```
kubectl get pods -l  'environment in (production),release in (canary)'
```

Kubernetes also includes a list of reserved labels in the `kubernetes.io` namespace. Some of these are applied to nodes by default and let us generate more useful queries using `nodeSelector` labels to manage resources.

For example, if we want to test a collection of low-power nodes in an edge cluster running on ARM SBCs, we can target them by running the following command:

```
kubectl get pods -l kubernetes.io/arch=arm64
```

Test Frequently

We should frequently test our entire set of Kubernetes pods and services. Running services with a collection of pods is good, but it requires that we test for performance, stability, and availability at scale. Load testing a node helps us see how pods interact under heavy workloads or behave on less powerful servers. We can identify what happens if the cluster loses a node and how our pods and services respond to this behavior.

We should also monitor every part of our Kubernetes infrastructure. We should start by ensuring our nodes have adequate observability, then proceed to more granular analyses of our pods, services, and containers. Where possible, we should rely on dashboarding tools and configure appropriate alerts and response plans to remedy component failures in our clusters — or better yet, avoid outages in the first place.

Provision Hardware

Although it's efficient to use built-in Kubernetes features for resource provisioning, it won't overcome a fundamentally hardware-based provisioning issue on its own. Right-sizing requires us to analyze the resource usage patterns for our workloads and build our infrastructure accordingly. Furthermore, to ensure resiliency, we should aim to have one or more spare nodes within each cluster.

The best tool for refining our Kubernetes practices and discovering the most optimal hardware arrangement is machine learning.

The Role of Machine Learning in Kubernetes Resource Management

Managing our resources in Kubernetes environments can be tedious, challenging, and time-consuming. In addition to keeping our clusters healthy, we also need to watch over pods, services, and containers, publishing them and validating them as needed. Frequent changes in our business workload requirements cascade into a continuous software workload. This means we're continuously updating software and juggling resource utilization to stay efficient.

The general progression towards automation has affected how we manage Kubernetes environments. Source-controlled YAML files and Helm charts present a good opportunity to navigate operational challenges by reliably automating our deployments.

A platform like **StormForge** extensively integrates machine learning (ML) into the automation process to help optimize resource management for our Kubernetes clusters.

The key to ML-based optimization can be summed up in a single word: data. It's the ability to contextualize the resource metrics (i.e. data) we can extract from a running Kubernetes environment so that our operations team can best fine-tune our cluster configurations in response.

ML lets us produce actionable intelligence in the form of optimal recommendations from the massive amounts of data our observability platforms generate. The insights are invaluable for optimizing our resource management and prove especially critical when we start to enter the realm of massive scale and its unpredictable effects on performance.

Using StormForge for Kubernetes Optimization

The StormForge platform uses machine learning to automate Kubernetes resource management for intelligent and efficient scaling. The platform approaches this holistically by using two techniques:

- Observation-based optimization
- Experimentation-based optimization

Observation-Based Optimization

Nearly 60% of organizations running Kubernetes consider infrastructure management to be their **top challenge**. The value of observability tools is limited when they're used in isolation. With the increasing complexity of most Kubernetes environments, the true challenge lies in taking action based on collected data.

So, how do we use the information we've collected most efficiently and effectively?

StormForge offers observation-based optimization via **StormForge Optimize Live**. This solution, an integral part of the StormForge platform, analyzes our existing Kubernetes observability data streams and offers CPU and memory recommendations to improve application performance. The optimizations can greatly reduce wasted resources in production environments.

Incorporating observation-based optimization into our Kubernetes resource management strategy is important because it empowers us to make changes based on accurate and timely insights — actual production data. We can use observation-based optimization to adjust our workloads in real time as they scale up and down.

With StormForge Optimize Live, we always have full control. We can choose to implement changes automatically or decide that recommendations should require manual approval.

How to Implement Observation-Based Optimization with StormForge

There are five key steps involved in this process:

1. We begin by creating a StormForge application, which is a group of Kubernetes resources that we want to optimize. Once we supply the namespace information, StormForge automatically identifies and displays the resources we can tune.

2. Next, we select which container resources we want to optimize. We also establish our level of risk tolerance for each application — for example, instructing StormForge to treat mission-critical applications more conservatively. Finally, we specify whether to deploy recommendations automatically or to require approval before deployment.
3. StormForge generates CPU and memory recommendations by applying its ML-based optimization engine to analyze performance and utilization data from Prometheus or other Kubernetes observability solutions. We can choose how frequently to receive these recommendations.
4. After StormForge finishes its analyses and provides recommendations, we implement them. Depending on our configuration settings, StormForge either does this automatically or allows you to review before deployment.
5. We can use the Grafana dashboard, shown here, to view immediate results from the analysis and optimization tasks.

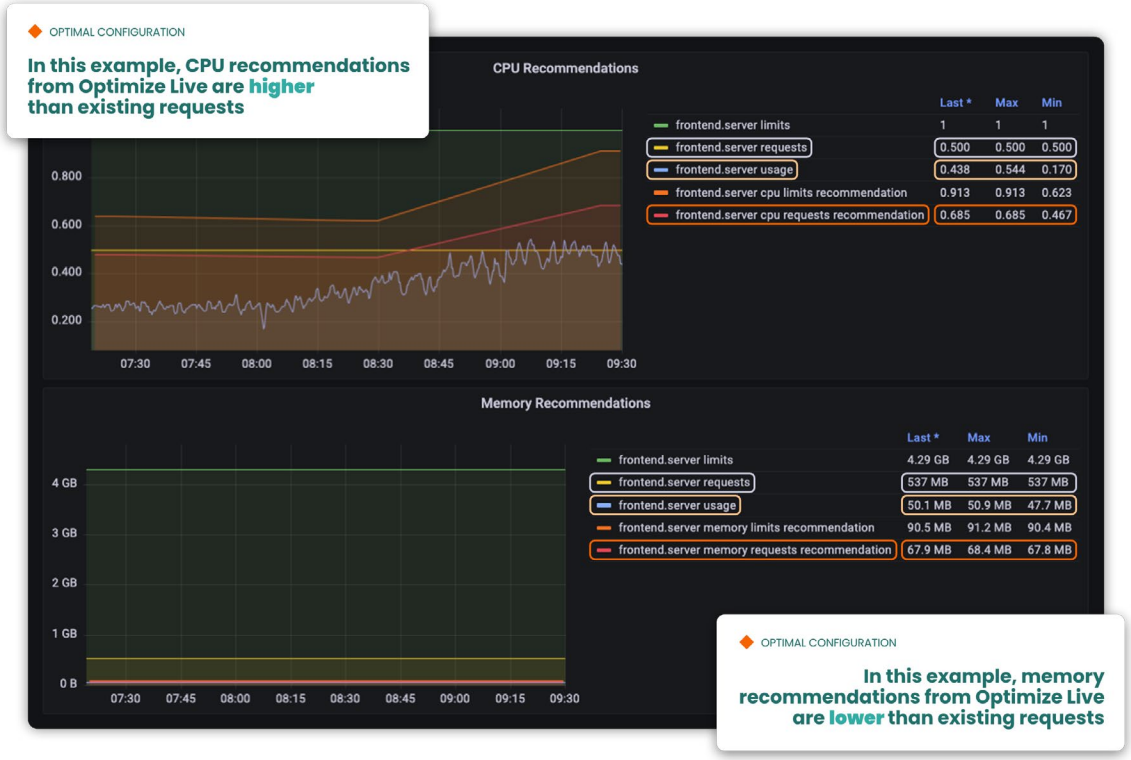

*Figure 2. StormForge provides recommendations for CPU and memory requests basesd on **actual** usage*

Experimentation-Based Optimization

Rather than analyzing existing data, experimentation-based optimization explores potential avenues forward when we don't have datasets for them. This method allows us to test a wide range of scenarios and ensure we've optimized before deploying.

StormForge Optimize Pro is an experimentation-based solution within the StormForge platform that provides fast, detailed application insights by running scenario analyses on non-production clusters. We can use StormForge Optimize Pro to simulate the effects of different configurations under hypothetical workloads, with several specialized features:

- Experimentation-based optimization in testing, development, and pre-production environments. StormForge Optimize Pro lets us run ML-backed analyses without affecting our business-critical workloads.
- Load testing scenarios and other simulations to test our system capabilities and help us clearly identify and adapt to our resource constraints.
- Highly configurable ML-based optimization that provides an extensive set of parameters to run different types of tasks.
- Deep application insights to drive architectural improvements in every part of our Kubernetes clusters.

Performance testing is another key component of a robust experimentation-based approach to optimization. Performance testing lets you quickly set up massive load testing for your app, scaling to hundreds of thousands of requests per second or millions of concurrent users with a setup that takes only minutes.

Simulating heavy loads is one of the best ways to analyze potential performance issues in our Kubernetes-backed apps before running them in production. StormForge includes **Performance Testing** as part of the platform, or you can use a different tool if you already have performance tests set up.

StormForge Performance Testing is designed with flexible and powerful features to ensure you meet your SLAs — and your end users' needs — are met:

- Setup is easy and takes only a few minutes.
- It offers tight integration between GitOps and DevOps by providing testing as code (TAC) and CI/CD pipeline scenario integrations.
- It's highly scalable, enabling simulations that scale up to millions of concurrent users.
- It's cloud-native, but also supports on-premises, hybrid, and public cloud architectures.
- It provides extensive reporting and analytics details, which simplify result interpretation, validating, and benchmarking.

StormForge Case Study: Acquia

Acquia is a PaaS company that offers a low-code, enterprise-level Drupal hosting solution. Acquia needed to optimize performance, scalability, and cost for its core hosting solution, the Acquia Digital Experience Platform. A key challenge Acquia faced was right-sizing a Kubernetes environment for each client — and for a range of application demands.

StormForge's intelligent scaling and optimization capabilities enabled Acquia to **forecast demand and make smart resource decisions** regarding application configuration. With continuous scenario planning available through StormForge Optimize Pro, Acquia was able to iterate through many more potential configurations than would have been possible to do manually. Through Optimize Pro, StormForge offered Acquia a bespoke solution that minimized cost while delivering the performance each client was looking for.

Acquia now runs StormForge Optimize Live to maintain this standard by using ML to analyze the data it already collects. Acquia delivers excellent value in a highly competitive market without the need for costly data aggregation or additional infrastructure maintenance.

Conclusion

Managing Kubernetes resources is a complex and tedious task, especially when provisioning and configuring for workloads at massive scale. To adequately manage large-scale containerized applications, we need to optimize their use of the underlying infrastructure in an environment with countless tunable variables and ephemeral components. Maintaining and refining healthy services and applications within our Kubernetes clusters requires that we consider the trade-offs between performance — our product's availability and scalability — and the cost of cloud resources. In most Kubernetes environments, unfortunately, these resources are over-provisioned and under-utilized.

The Kubernetes framework inherently offers some capabilities for automated resource management and scalability. However, automating a process that doesn't appropriately analyze resource behavior is equivalent to automating a bad process, which ultimately yields suboptimal performance, scalability, and cost-efficiency — and a poor user experience. By applying machine learning-based solutions like those on the StormForge platform, we can optimize our automations based on data derived from both observation and experimentation. This transforms our data into actionable intelligence that delivers superior results for both production and non-production Kubernetes environments.

Request a demo to learn more and experience how StormForge can optimize your Kubernetes environment.

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