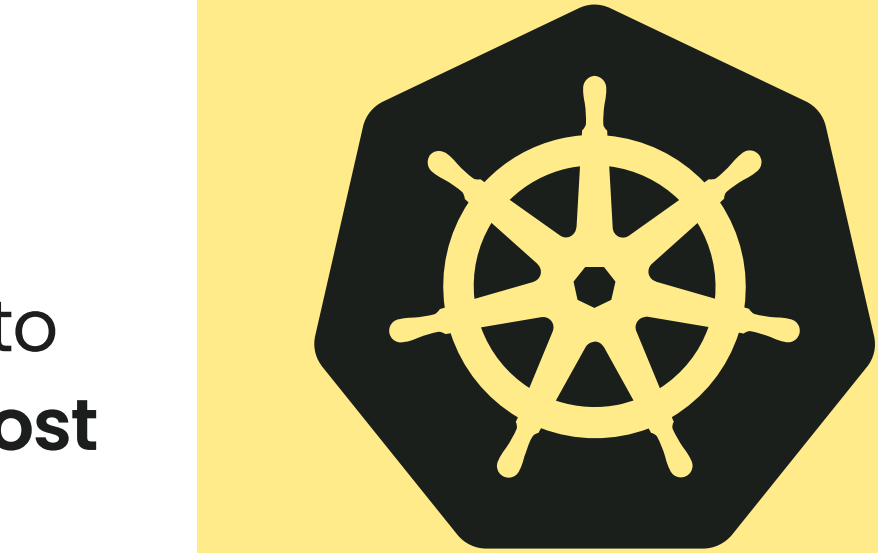
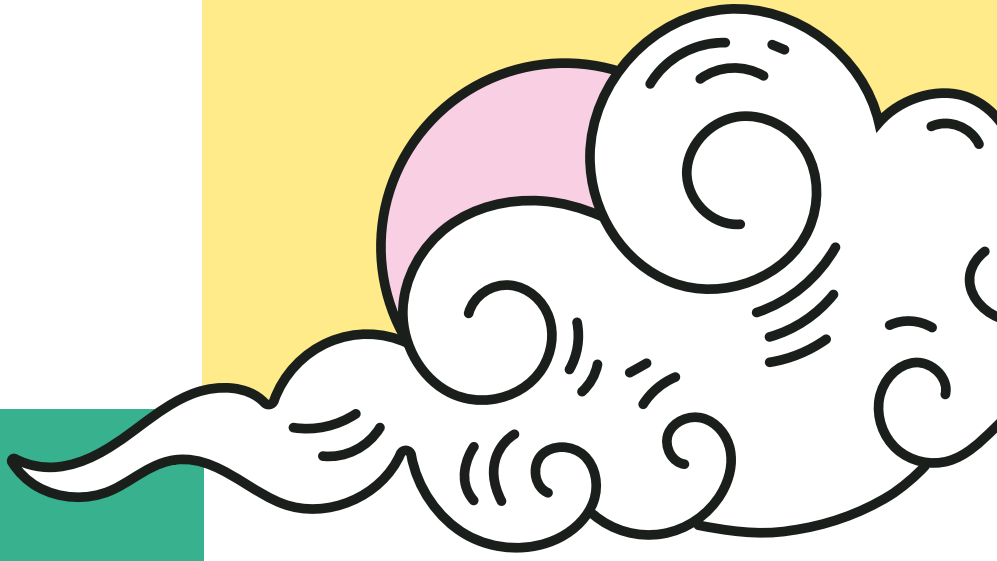
eBook

The Essential FinOps Guide to **Kubernetes Cost Management**





finout

Kubernetes – The go-to platform for container management.

Table of content:

Introduction

Businesses are deploying more and more containerized applications to the cloud. This has, naturally, led to an increase in the use of Kubernetes – the de facto container management platform. K8s enables engineers to optimize clusters for cost, performance, resilience, capacity, and scalability – by proactively creating, supporting, and dumping container instances as needed.

However, cost management was never one of Kubernetes' core features. While containerization is a sublime solution to the issue of scalability, it certainly removes the likelihood of a "fixed cost" for your cloud usage by day, let alone by month. Add to this the complexity that arises from multi-tenancy architecture, and many businesses suffer an ongoing struggle to assign costs and forecast spending accurately.

This guide will lead you through the process of recognizing and addressing the challenges associated with achieving cost observability in Kubernetes-orchestrated applications, detailing step-by-step solutions you can implement.

Let's start by examining why determining the cost of running distributed applications in the cloud can be such a challenge.

Challenges of Kubernetes cost observability

When managing Kubernetes, controlling the cost of resource consumption is difficult for both startups and established organizations alike. It's particularly confusing and complex to analyze the cost incurred by each pod, and developers tend to play it safe and over-allocate resources, resulting in spending outstripping use.

Consider the mechanics of the micro services architecture layer: the Kubernetes control plane distributes applications as pods to worker nodes and runs them. Worker nodes are the virtual machine (VM) instances where actual Kubernetes workloads run. It is possible to specify the resource requirements of pods such as CPU, RAM, and storage – and these resources are supported by different VM instances, each with its own price tag. If you use a cloud provider such as AWS, you are probably also implementing Amazon EC2 instances as computation nodes for pods in your Kubernetes cluster.

With multiple applications running on the same worker nodes and being scaled up and down automatically, it's fairly complex and time consuming to calculate the actual, or estimated, cost of an application running in Kubernetes.

Let's clarify this with a real-world scenario. Assume you've deployed a MySQL database to your Kubernetes cluster on AWS. Kubernetes not only starts a MySQL container in one of the nodes in your cluster, but it also creates volumes, secrets, stateful sets, replica sets, pods, service account users, configuration maps, services, and ingresses to make your new MySQL database scalable, reliable, and reachable.

Yes, those costs for the infrastructure items will be listed in the AWS Cost and Usage Reports (AWS CUR), such as storage or compute. The simplest method to gain visibility over cloud costs is to apply tagging to resources based on their projects, owners, and

other metadata. However, with a dynamic orchestrator like Kubernetes, it is not feasible to tag every single Kubernetes resource in the cluster. There is no simple way to break the abstraction provided by K8s API and attribute the cost of the cloud infrastructure to K8s resources such as your MySQL.

Figure 1. Cloud cost bills reflect your provider's unit of economics, not yours!

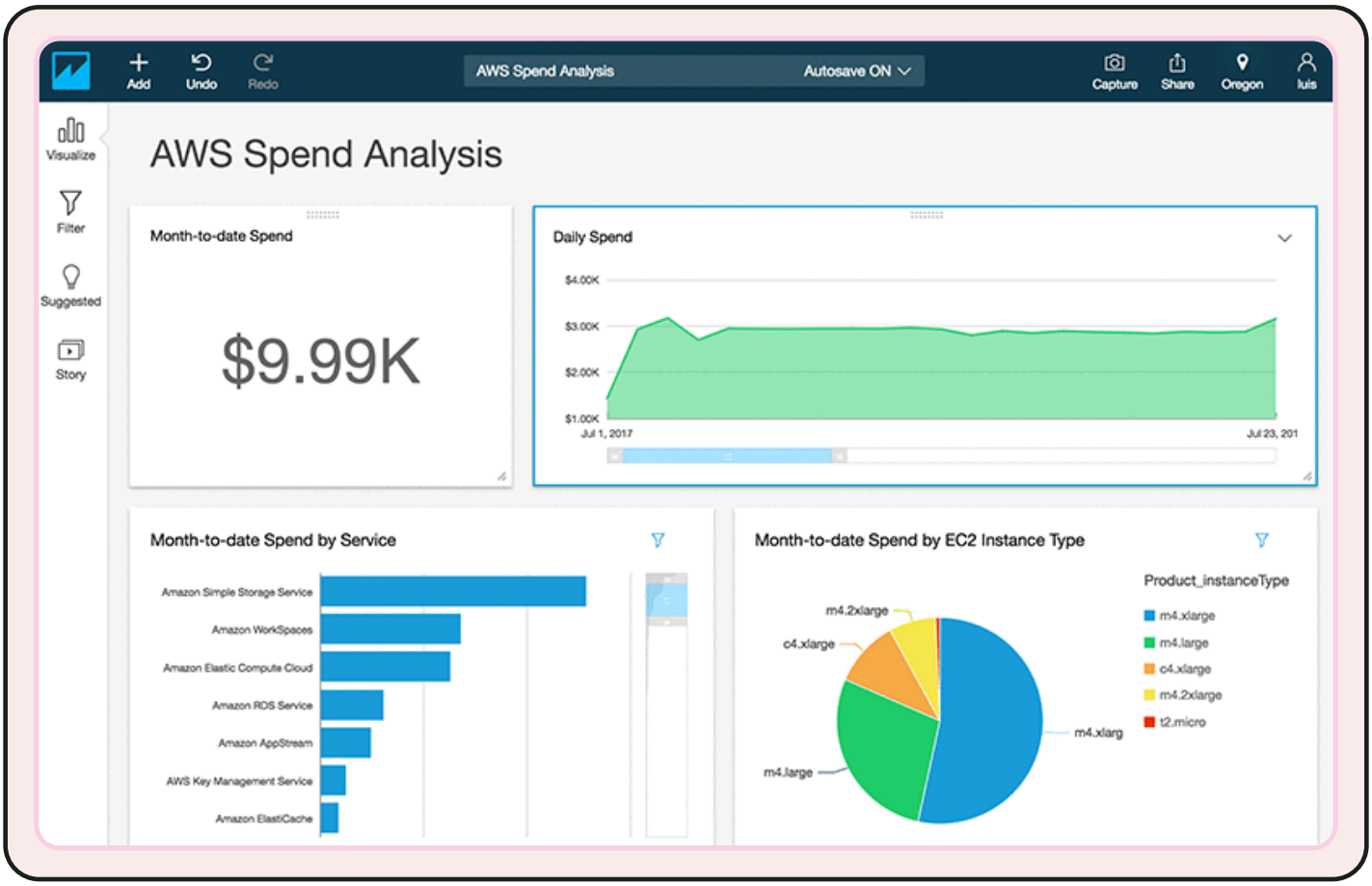


Image source: [AWS blog]

There are two critical points to understand regarding your cloud bill. First, the billing unit is not the same as the unit of economics that you actually apply to your business. Your cloud bill shows the unit costs according to metrics that matter to the cloud providers themselves (Figure 1), not to you. Second, billing statements are not granular enough to allow you to genuinely understand the fees associated with supporting your application/s.

Know your enemy

As noted earlier, developers tend to overestimate resource allocation. It's also beneficial to recognize other common challenges when strategizing on how to optimize your Kubernetes spending.

Resource asymmetry

Typically, a K8s ecosystem supports multiple workloads with different resource requirements. Some applications rely on high CPU, while others are memory intensive. Such resource asymmetry is handled by the kube-scheduler, which applies a scoring algorithm to assign the right node for each workload. The scoring algorithm is often a combination of multiple factors, and comprehending how and when a node is to be allocated to a workload can be complex.

Fluctuating resource consumption

Another crucial factor in controlling consumption costs is the varied usage of resources by Kubernetes workloads. As CPU and memory resources constantly change for dynamic workloads, every change in resource consumption is responded to by the fresh provisioning, realignment, or destroying of pods within nodes.

Missing requests and limit

Request and limit parameters help optimally allocate resources to pod workloads within a node. In a node with multiple pods, if one among them does not have these parameters defined, this pod tends to consume all resources available in the node. In such instances, other pods never get the resources they require, impacting the overall performance and cost of the cluster.

These challenges, and more, can be overcome by applying the best practices considered within this guide. Before we address the how, let's consider what our objective is.

Know your destination

Part of the power of the Kubernetes API is the strong abstraction it places between the infrastructure and applications running on a cluster. However, the challenge then becomes: Which Kubernetes resource should be allocated the costs?

The task becomes identifying a Kubernetes unit for cost calculation such as pods, deployments, or namespaces. Whichever you choose, the outcome should be that you have visibility over the cost of your dev namespaceor test deployments.

Before we dive into the nitty-gritty of setting up Kubernetes to improve your cost observability, we will take a good hard look at a cloud financial management discipline and cultural practice that enables organizations to get maximum business value from the cloud: FinOps. Because there is no point in collecting data if you don't have the structures in place to make meaningful adjustments in light of the information it provides.

FinOps

FinOps (short for Financial Operations) is an important cloud operating model that enables cross-functional teams from technology, business, and finance to work together. Applying FinOps binds different stakeholder groups – thanks to the shared language and processes. This enables engineering and finance teams, business stakeholders, and executives to manage your cloud costs better and deliver business features faster. A FinOps implementation goes beyond simple cost-saving strategies. In fact, it is often vital to increase your cloud spend to drive more revenue or to support the growth in your customer base. FinOps isn't just about tracking cloud costs; it's also about making decisions that will increase the cloud's business value. FinOps empowers teams to be agile against blockers and respond to opportunities to optimize their spending.

This section will take you step-by-step through understanding the challenges of creating cost observability when implementing K8s-orchestrated applications and the solutions you can apply.

Let's start by examining why determining the cost of running distributed applications in the cloud can be such a challenge.

FinOps principles

The FinOps foundation offers several guidelines to help your organization adopt a successful FinOps practice.

Collaborate across teams

Applying FinOps often requires a cultural shift to ensure that teams no longer work in silos. FinOps increases collaboration between teams and can help your organization lay a solid foundation for your processes around cloud financial management.

Make decisions based on the cloud's business value

Metrics provided by cloud services offer many potential benefits to developers and can help them enhance the delivery of their applications. Development teams can consider cost in the same way as they do other efficiency metrics to incorporate it as a non-functional requirement and improve the delivery of business features.

Form a centralized FinOps team

The FinOps team takes responsibility for your cloud strategy and cost governance. They must set cost-efficiency expectations and educate engineering teams with regard to best practices and standards. By establishing cloud cost-control guardrails and cost-tagging standards, the FinOps team creates budgets and cost forecasts, develops dashboards, and works to optimize organizational costs.

Take ownership of cloud usage

FinOps is only functional when each team in your organization takes responsibility for the cloud costs of its products and resources. By tracking team-level expenses and providing in-depth visibility across all spending, teams are empowered to optimize costs. Providing data is key to stimulating ownership over workload-level cost efficiency and enabling rightsizing exercises.

Create easily accessible FinOps reports

Ensuring that everyone has access to cost reports is a key step to managing cloud spending. Access to cost dashboards supports forecasting and simplifies tracking the cloud usage of applications. Sophisticated reports implement anomaly detection, ensuring that teams are immediately notified of an abnormal event. This is vital to setting up a fast feedback loop.

Take advantage of the cloud's variable cost model

It is perhaps a little too easy to provision new resources in the cloud. This means that engineers need to remain cost-conscious and ensure optimal resource usage. Savings are often identified by undertaking cost optimization practices like rightsizing resources, purchasing reserved instances, and turning off resources when not in use or in development/sandbox environments.

The FinOps lifecycle

These core FinOps principles go hand-in-hand with three distinct phases – inform, optimize, and operate. These should be continuously cycled through.

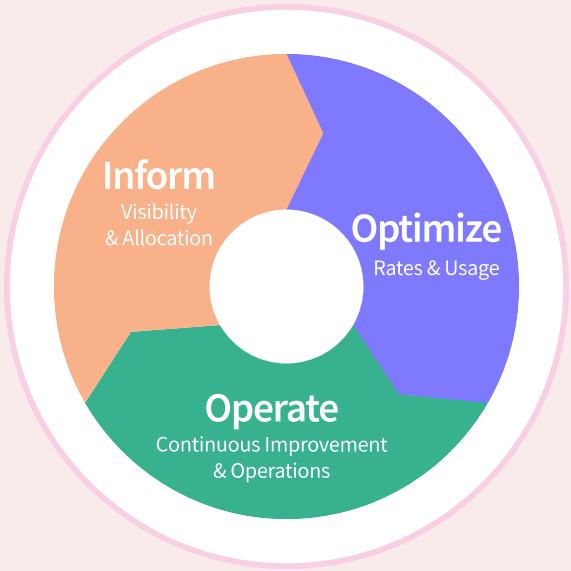


Image source: FinOps foundation

Inform

The inform phase provides teams with visibility into their cloud spending in near real-time- assisting with a granular understanding. It requires mapping costs to their applications and business units. These can then be analyzed and rolled up in reports to generate budgets, forecasts, cost dashboards, and scorecards.

A key component of this phase is to implement a standardized tagging strategy. This step is crucial to provide visibility into usage and spending. Untagged resources must be identified and tagged to ensure full cost transparency and accurate chargeback. With this phase initiated, your FinOps teams are immediately better informed about costs and able to identify potential areas for optimization.

Optimize

Your cloud vendor's cost governance and optimization tools can help you review your organization's costs and identify trends in usage data via an interactive interface. Such tools can identify underutilized resources – assisting you to eliminate any resource wastage. You may be able to identify opportunities for purchasing reserved instances or create a savings plan to lower long-term costs. It is also important to compare options for expensive resources against similar third-party or cloud vendor's services.

Setting up cost anomaly alerts is an excellent strategy to identify any unusual usage patterns or cost spikes. Alerts assist the FinOps team in analyzing your entire fleet of resources.

Operate

The operation phase is when you implement your cost optimization plan created in the optimization phase. FinOps team members do not implement the changes; rather, they provide guidance and visibility into cloud usage. Ensure engineering teams are both trained in effective cost optimization strategies as well as empowered to implement the recommendations. Ultimately, it is your engineering teams that make those infrastructure changes to optimize their cloud spending.

Never take your eye off that ball!

Continuous improvement and automation are both important when building an effective cost governance strategy. It is important to support automation in your processes so that these cost optimization measures can be performed repeatedly. Stakeholders should regularly review cost reports to ensure that decisions are timely and to minimize the feedback loop.

Now that you understand FinOps principles and the FinOps lifecycle, we can take a detailed look at the specifics of gaining visibility into costs associated with Kubernetes cloud environments.

Implementing Kubernetes cost observability

Gaining visibility into your Kubernetes costs in the cloud involves a dual approach:

Step 1 – Track the actual usage: Your public cloud billing reports, such as AWS CUR, will provide the total CPU, memory, networking, and monthly storage cost. It is then possible to allocate CPU and memory according to the Kubernetes cost monitoring resource limits or actual usage metrics.

Step 2 – Distribute shared expenses: Those shared expenses such as networking and storage must be assigned to particular projects, teams, namespaces, and applications. Also, suppose your autoscaled architecture includes pods that support a multi-tenant service. In that case, you need to map these Kubernetes units for cost calculation – such as pods, deployments, or namespaces to the abstraction layer – the unit of economics. By creating such methods to allocate costs, FinOps teams can zoom in on single-tenant, single-team, and single-application costs.

There are several strategies that you can apply to assist with tracking to enable better expense allocation in Kubernetes, and it is these that we will consider next.

Step 1 tracking: Apply labels in Kubernetes

Labels are a key control layer supported by K8s. Most people probably first use labels to optimize how they leverage K8s simple API and third-party integrations. For example, labels allow your cluster to communicate with client tools and libraries such as kubectl and Helm.

However, the application of Kubernetes labels can be far more nuanced than this single use case. If Engineering needs to debug an issue, DevOps wants to shut down non-essential infrastructure resources over a holiday, or FinOps wants to understand the costs in a multi-tenant environment in K8s, labels give you that power.

To apply labels, you first need a firm understanding of:

- Namespaces
- What labels are
- How to apply them
- How to conduct searches and the benefits they offer

What is a Kubernetes namespace?

A namespace is a high-level collection of K8s resources. They are intended to simplify the management of resources in environments where many users are spread across multiple teams or projects. A resource's name must be unique within a namespace (but not across namespaces).

Because each Kubernetes resource can only belong to one namespace, they create natural divisions between teams and applications. Using the ResourceQuota object, each namespace can be allocated with the number of objects and a fair share of the Kubernetes cluster's resources, including:

- Memory limit
- Memory request
- CPU limit
- CPU request

Breaking down a cluster into multiple namespaces and monitoring each namespace helps identify which team or service has more cost overhead and assists with optimization efforts.

Your team may not need to apply namespace names. In fact, for clusters with user counts up to a factor of ten, you will probably not need to create them at all. The reason they are important to our discussion is that labels are also used to distinguish resources within the same namespace.

What are Kubernetes labels?

K8 labels are key-value pairs that are part of an application's metadata. Kubernetes labels offer a simple technique for identifying Kubernetes objects and aligning them into groups.

With Kubernetes, the concept of an application is (deliberately) left very open and defined with metadata. This means that creating your own label is a very open-ended process. This leaves it entirely within your control what data you choose to hold.

That is not to say there are no conventions. In fact, the Kubernetes services use labels to schedule pods to nodes, manage replicas of deployments, and network routing of services.

Labels or annotations?

Two properties may hold key-value pairs, labels and annotations. Labels should be used to attach identifying metadata, for example:

```
{
  ...
  "metadata": {
   "labels": {
    "tenant": "explo-6834",
    "environment": "production",
    "tier": "backend",
    "app": "#5784762",
    "version": "1.82"
   }
  }
}
```

Annotations are used to attach additional arbitrary data to objects. For example:

```
{
  ...
  "metadata": {
    "annotations": {
      "first_deployment": "1646126616",
      "deployed_by": "daena@example.com"
    }
  }
}
```

Note, if you have set up an annotation that later becomes used to group objects, you probably need to consider that this has been elevated to a "label", and reassign it.

Kubernetes standard labels

Kubernetes services and replication controllers use labels to manage pods, target workloads to specific instance types, and control services across multiple cloud provider zones. Therefore, label use is hard-baked into the Kubernetes design.

Standard labels include:

key:	Pair (the property's value)
app.kubernetes.io/name:	Name of the application
app.kubernetes.io/part-of:	Name of the application
app.kubernetes.io/managed-by:	Name of the application

Many standard labels are auto-filled by K8s, so it is well worth applying them for your daily operations and client tools. **For example:**
`app.kubernetes.io/managed-by: "" `

Will be populated with:
`app.kubernetes.io/managed-by: helm`
if “helm” is the package manager.

Why setup custom Kubernetes labels?

Custom labels are an excellent solution to several challenges that you will face when setting up a Kubernetes environment. They are very similar to the tagging concept in AWS: AWS tagging also relies on key-value pairs that identify AWS resources in EC2, S3, Redshift, and EFS.

Kubernetes labels allow DevOps to optimize searches, apply configurations, and manage deployment administration. Labels also enable FinOps by implementing a cost monitoring mechanism by identifying the pod-level resource usage for different environments or applications.

Custom labels in action

Say that DevOps wants to monitor the status of pods according to their environment; you can set up key pairs that identify the environment, such as:

environment: development
environment: staging
environment: production

Such granular data allows you to make specific calls. For example, you want to list the status of all production pods:

```
...
kubectl get pods -l 'environment=production'
...
```

This is far superior than having to make an API call for all pods and then filtering through the output after.

One of the benefits of applying a labeling strategy is that although each component may be granular, labels collate them into the bigger picture — letting you zoom into or rebuild that picture. Let's say you have a multi-tenant environment. If you want to know all of the services a particular tenant uses, then you can collate that tenant's data just by filtering.

Say our example tenant:

"tenant": "explo-6834"

is supported on tier==backend and tier==frontend.

Should you receive a query with regards to a perceived service issue, a simple API call retrieves all the service-related data you need for that tenant. No need to look up any system diagrams to see what applications support that tenant's service. You retain your modularity without losing any visibility.

Labels are also very useful for release management. Found a backend bug and want to release a patch? Simply deploy a new set of v:1.83 backend instances, replace tier:backend, version:v1.82 with tier:backend, v:1.83 in the service label selector. The pods running v1.82 were orphaned, and you have deployed a new set of instances.

Constraints on labels

The following syntax constraints are applied to labels:

- Key must be unique within a given object.
- Min 0-max 63 characters for the segment (required): 253 for prefix (optional).
- Start and end with alphanumerics [a-z0-9A-Z] (unless length is 0).
- Dashes "-", underscore "_" and dot "." allowed (internally).
- (Optional) prefix must be a series of DNS labels separated by dots and followed by a slash.

The inclusion of the prefix allows users and automated system components, for example, kube-scheduler, or third-party integrations, to manage resources.

Let's unpack those two syntax constraints that could cause confusion a little further: Enforcing the key as unique prevents us from making copy/paste mistakes such as duplicating the environment property.

```
{
  ...
  "metadata": {
      "labels": {
         "environment": "production"
      }
   }
}
```

- Consider a standard label such as: app.kubernetes.io/name.
- {app.kubernetes.io} is the prefix providing the DNS label.
- {name} is the segment.

Searches

The Kubernetes API supports searches for:

- Equality, i.e., 1:1 matches.
- Inequality, i.e., specify a "does not match".
- Sets.

Equality uses = (or, if the fear of resetting a value leaves you feeling itchy, ==). Inequality is the standard !=, and a set or array of values is specified with a comma separator.

From our previous example of labeling our environment, therefore, we could use:

- Equality, to return the data on the pods in production:

```
...
kubectl get pods -l 'environment==production'
...
```

- Inequality to return the pods in production and development

```
...
kubectl get pods -l 'environment!=(staging)'
...
```

Or we can search for sets, i.e., an array. Set searches apply "in", "notin", and "exists":

```
...
kubectl get pods -l 'environment in (production)'
...
```

to return the data on the pods in production. Or

```
...
kubectl get pods -l 'environment notin (development, staging)'
...
```

Where the separating "," comma acts as an AND (&&) operator.
Note that OR (||) is not supported.

Multiple conditions

If you provide more than one equality condition, then the matching object/s must satisfy all of the constraints. For example:

environment=production
tier!=frontend

Similarly, set-based conditions return the sub-set of objects that match all the given conditions.

Thus, according to our example:

```
{
  ...
  "metadata": {
   "labels": {
    "tenant": "explo-6834",
    "environment": "production",
    "tier": "backend",
    "app": "#5784762",
    "version": "1.82"
   }
  }
}
```

Where the environment may be: staging, development, or production, then:

```
...
kubectl get pods -l 'environment in (production)'
...
```

would return the same object as

```
...
kubectl get pods -l environment=production,tier!=frontend
...
```

Labels: Best practices

There are three key considerations when implementing best practices for your K8s labels:

1. Have a labeling strategy
2. Use templates
3. Automate(CI/CD) labeling

① Your organization labeling strategy

When a system is designed to be open-ended, it is important to apply your own strategies and conventions to ensure that your labels provide you with the functionality you need. Once such conventions are established, you can add checks at the Pull Request (PR) level to verify that configuration files include all the required labels.

Setting an informative prefix can assist you in instantly identifying which service or family of functions a label applies to. It is good practice to choose a prefix to represent your company and sub-prefixes to identify specific projects.

If you want to see the labels applied to an object, you can add this flag to your call:

```
...
kubectl get pod my-example-pod --show-labels
...
```

② Templates

In K8s, the concept of a template has a very specific application, thanks to pod templates. However, it used to be that the term “template” meant "a shaped piece of rigid material used as a pattern for processes such as cutting out".

So, let’s apply the term beyond pod templates, because it is good practice to apply a rigid structure to shape all your configuration files with ready-to-use patterns. From `PodTemplate` (specifications for creating pods) to metadata structures for applications, your team will all be on the same page with a ready-to-use label strategy in place.

What you tag will depend on your needs, but will probably include those provided in our examples, such as:

- Environment
- Tier
- Version
- Application uuid

And, in a multi-tenant environment, where a pod is dedicated to one tenant, never forget:

- Tenant

Because you will love the cloud cost management K8s provides you by including tenancy!

It is recommended that you ensure your map labels for all business units incurring a cost so that they may be aligned with consumption logs for shared resources. This makes cost analysis, reporting, and optimization easier for individual teams, services, or business lines. Once you have defined a labeling strategy that teams can apply, the next best practice step is to validate the process. Conduct static code analysis of all resource config YAML files to verify the presence of all required labels. A PR should only be merged if the configuration file provides all the required labels.

③ Automate labelling for CI/CD

Within your continuous integration/continuous delivery (CI/CD) pipeline, you can automate some labels for cross-cutting concerns. Attaching labels automatically with CD tooling ensures consistency and spares developer effort. Again, validate that those labels are in place: CI jobs should enforce proper labeling by making a build fail and notifying the responsible team if a label is missing.

You can define variables on Jenkins files or GitHub actions workflows and parameterize the label part to automate labels in Kubernetes manifests. At the same time, you can use Helm to easily deploy each version you want. And automated labels will help you here for each deployment strategy (canary, rolling update, etc.). **Helm sample usage:**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:
  labels:
    app.kubernetes.io/name:
    app.kubernetes.io/instance:
  annotations:
    kubernetes.io/change-cause:
```

Step 2- Distribute expenses: Apply the unit of economics

It is clear that, with labeling, cost insights can be relatively simple to achieve if you are in the enviable position of being able to assign a Kubernetes namespace to each tenant or project. In reality, DevOps usually faces the challenge of measuring a tenant's usage of shared, autoscaled resources — which makes cost allocation more complicated. Cost allocation often requires assigning a tenant's pro-rata usage of the cluster's resources, including CPU, GPU, memory, disk, and network. This is where an abstraction layer, the Unit of Economics, comes in — to let FinOps zoom in on single-tenant or single-project costs.

Use cost monitoring tools to simplify FinOps

Cost analysis, reporting, and optimization are simplified by applying cost monitoring tools. Since cloud-native tools offer limited features for tracking expenses, companies often adopt third-party cost management tools to forecast budgets and monitor cluster expenses. These tools provide real-time visibility of pod-level resource utilization and offer insightful reports about the expenses incurred. Automation is one of the most crucial features that such tools offer, helping to detect, analyze, and report any resource usage abnormalities before they adversely affect budgets.

Having set up your tagging with labels, you began phase 1 of your FinOps journey: Inform. You are now able to identify over-provisioned resources and map labels for all business units incurring a cost against consumption logs for shared resources. It is that coherent labeling strategy that will now pay out dividends into your FinOps strategy, **allowing you to:**

- Identify over-provisioned resources.
- Allocate cost to individual teams, applications, or services. •
- Adjust cost by individual team, application, or service.

Using your cost monitoring tool empowers you to undertake phase 2 of your FinOps journey: Optimize and take informed optimization decisions. Some popular cost-monitoring tools include Kubecost, Prometheus, Kubernetes Dashboard, the ELK Stack, and of course, Finout. Let's take a look at these in more detail.

Cloud cost monitoring tools

Kubernetes does not come with an out-of-the-box cost observability tool, which is why many teams implement cloud cost monitoring. Such tools simplify the process of understanding and calculating the costs of running your applications in the cloud.

Selecting cloud cost monitoring tools

When selecting such a tool, what criteria should you be considering? We recommend measuring potential solutions against the following five performance criteria:

1. Installation: It should be easy to install and set up any tool from a third-party provider. Cost observability tools must also be installed with minimal intrusion on the cluster to limit performance/security concerns.
2. Ease of configuration: The Kubernetes API creates an abstraction layer between the infrastructure and end-user applications. It is critical that your cost monitoring layer supports configuration options that assist in breaking the abstraction layer. A faulty configuration will affect budgets due to mistaken calculations and estimations.
3. Granular cost visibility: One of the critical characteristics of cost observability is to provide a fine-grained view over costs such as costs per pod/deployment/namespace, and other resources in your cluster.
4. Connection to external billing (such as AWS billing data): Your Kubernetes monitoring tool should connect to your cloud provider's billing system. Your AWS bill, for example, has cost information that is essential to accurate Kubernetes cost.
5. Open-source: Kubernetes is an open-source platform and its popularity stems at least in part from this. Check the license of the tools you are using. A willingness to work open source is always a good trait in a potential service provider.
6. Open-source: Kubernetes is an open-source platform and its popularity stems at least in part from this. Check the license of the tools you are using. A willingness to work open source is always a good trait in a potential service provider.

The cloud cost monitoring tool ecosystem is growing as new service providers attempt to gain traction in the field. We summarized how the top five trending cost observability tools measures up against these criteria in Table 1.

Table 1. Top 5 Cloud Cost Monitoring Tools: Performance Assessment

	Configuration dificulty	Installation	Cost visibility	Connection to external billing	Open-source & community-driven
Finout	Easy	Easy	Yes	Yes	Partly ♦
Kubernetes dashboard	Easy	Easy	No	No	Yes
Prometheus	Hard	Hard	No	No	Yes
ELK stack	Hard	Hard	No	No	Yes
Kubecost	Easy	Easy	Yes	Yes*	Yes

* Identify over-provisioned resources

♦ Allocate cost to individual teams, applications, or services •

Cloud cost monitoring — your options

Let's take a closer look at what you can expect from these tools:

1. **Finout |** Finout is designed from the ground up with FinOps in mind. With a straightforward configuration and installation, it integrates with your Prometheus DB using Finout's open-source cronjob or with your Datadog account API. Either approach provides access to your Kubernetes cluster metrics (CPU and memory).

   Once integrated, Finout has access to cost per pod, deployment, namespace, cron job, StatefuSet, and cluster; using these metrics to enrich your AWS billing data with granular cost visibility. Once this level of cost visibility is achieved, the problem of allocating cost per customer/tenant/dev team/business application is solved. Using Finout, the user can quickly and accurately report, via a simple and intuitive platform, which K8s and AWS components each business unit is composed of and their costs — just as if Amazon sent you the bill.

② **Kubernetes dashboard |** Kubernetes Dashboard is an open-source, general-purpose web UI for Kubernetes clusters. It is part of the official Kubernetes landscape, so its configuration and installation are straightforward. With Kubernetes Dashboard, you can check what is running in your cluster and see its distribution to your worker nodes. However, the dashboard does not provide any information about cost visibility, such as price per pod or deployment. In addition, you cannot connect the dashboard to any external billing system, such as AWS Billing, for data collection or enrichment.

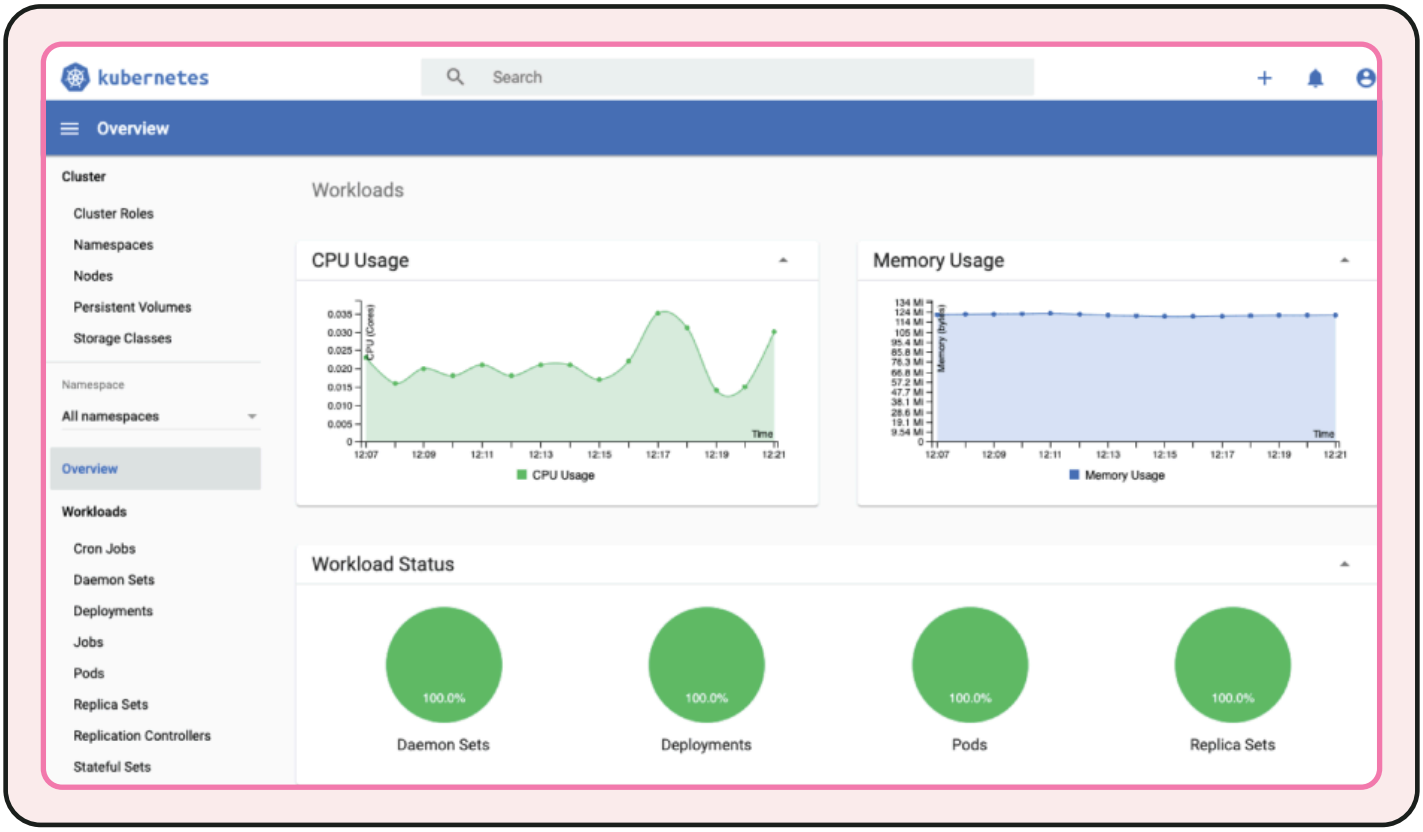


Image source: Kubernetes

③ **Prometheus + Grafana |** Prometheus is today's leading open-source monitoring framework that provides out-of-the-box monitoring capabilities for Kubernetes. However, it is neither easy to install nor configure. Configured properly, Prometheus can collect CPU, memory, and storage metrics from all your pods and nodes — in addition to Kubernetes-specific metrics. However, it lacks the cost visibility for Kubernetes resources and a connection to external billing systems.

By the way, you can integrate Prometheus with Grafana (multi-platform open source analytics and interactive visualization web application) and increase the visibility of your collected metrics.

Image source: Kubernetes

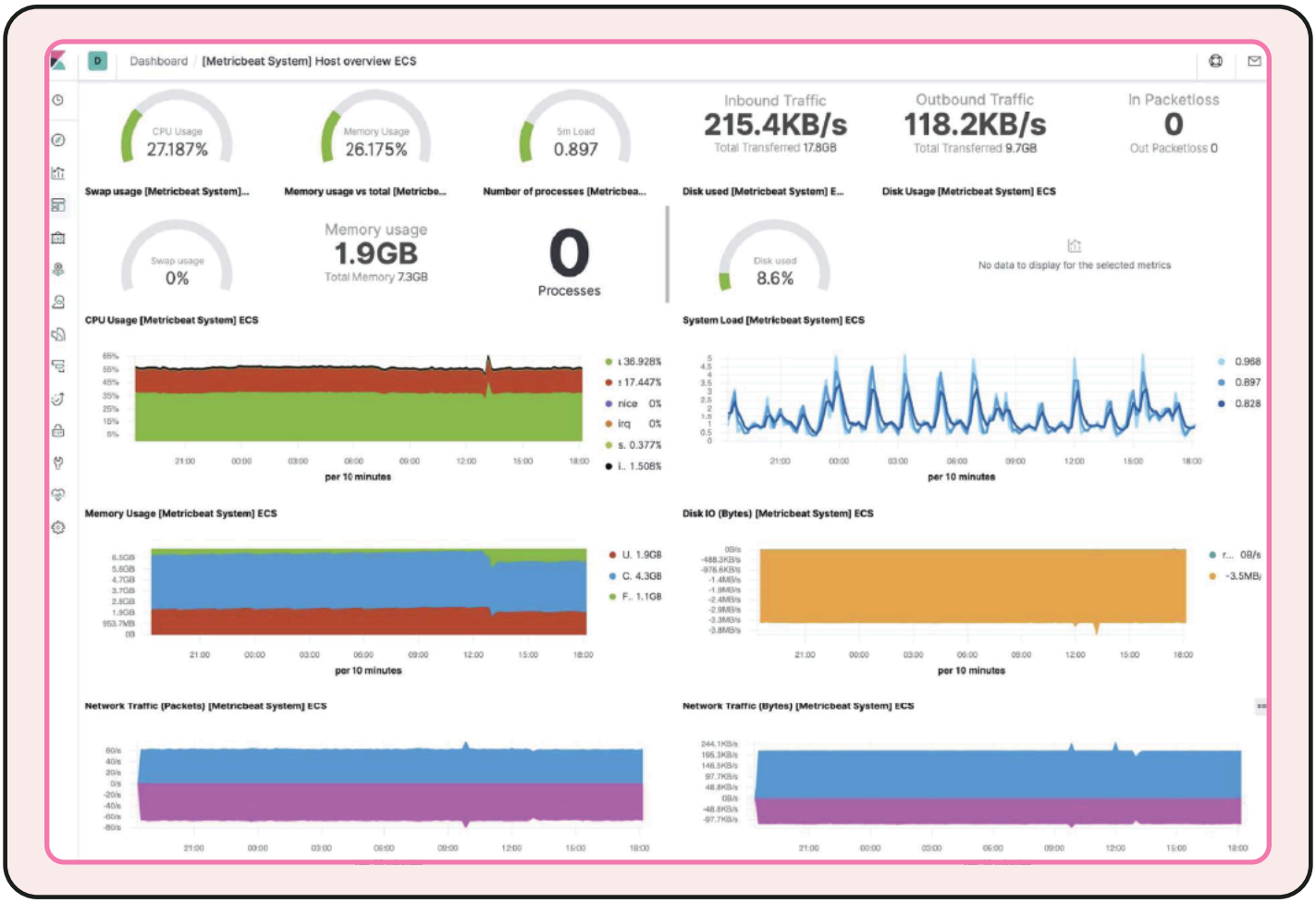
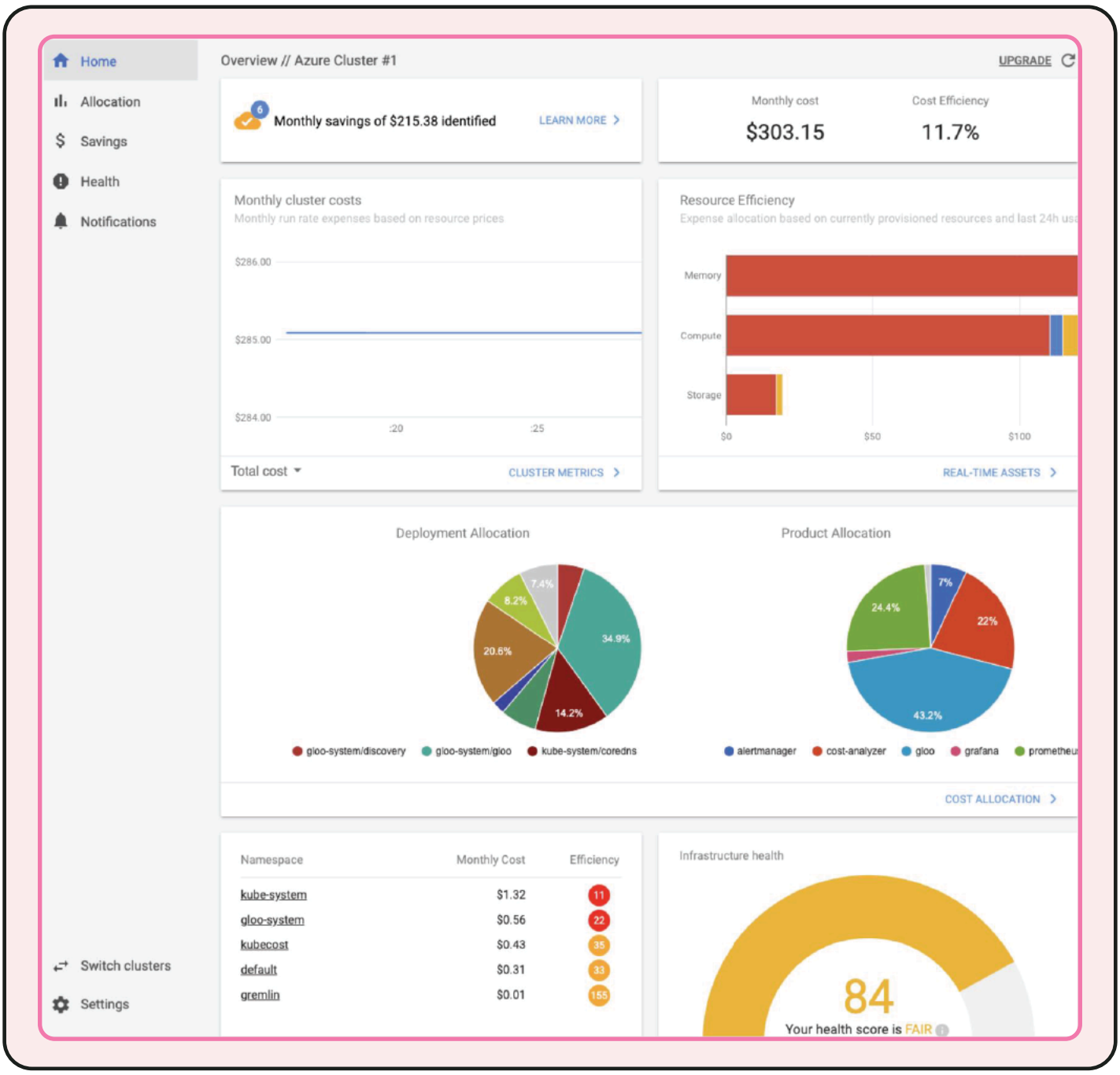


Image source: Kubernetes

④ **The ELK Stack |** The ELK Stack is a collection of three popular open-source tools: Elasticsearch, Logstash, and Kibana. The stack provides a central location to collect, analyze, and view logs of applications running in the cluster. Because of this, it is a valuable option for diagnosing and troubleshooting the problems of distributed applications. However, it is difficult to install and configure, and lacks a connection to external billing systems or cost visibility over Kubernetes resources.



5. **Kubecost |** Kubecost is a cost monitoring and management tool that focuses on cost visibility and cost control. Kubecost's free offering is an open-source solution with basic configuration and installation into the cluster. It helps reveal the actual price of your Kubernetes resources, such as pods or deployments. However, you can't connect it directly to your cloud cost management tool external billing systems, such as AWS Billing. In order to enrich and consolidate data, you need to connect your billing data to an AWS Athena database. In addition, most of its major capabilities are locked behind a paywall.

Cloud cost monitoring tools are not simply about spending less; after all, to grow, you need to scale. Often though, such tools can be used to protect those budgets and identify overspends. Let's finish off by taking a look at budget-saving initiatives that you can undertake to optimize your spending in K8s.

Optimize your Kubernetes spend

By gaining better visibility over your Kubernetes costs, you have achieved phase 1 of the FinOps lifecycle “Inform”. Now informed, it is time to Optimize. And, with a service as flexible as Kubernetes, there are several optimizations that can improve your bottom line.

Rightsize

Rightsizing pods is one of the most effective approaches to limiting budget overruns. **Define, measure, and update the resource requirements of pods.** Identify pods with surplus resource allocation to optimize and free up resources.

Based on the value of the request and limit pod parameters, Kubernetes scheduler can choose the node for the pod. This allows the pods to function normally, ensuring better stability and less resource wastage. Defining the right values for the Limit ensures that the pod is only assigned the amount of resources requested by the Limit and prevents starving other pods due to resource unavailability.

As a pod can contain multiple containers, you should set the parameters for all containers to get the aggregate request and limit that the pod requires.

Right Zone

Distribute workloads to cheaper regions, zones, and nodes. Pricing is not the same for each region, zone, or server even within one provider, for example, per AWS cost management provider. Always consider moving some of your applications to other parts of the cloud to optimize costs.

Right time

Create a mix of different nodes and use an optimization strategy for scheduling: Kubernetes can assign workloads to particular servers by grouping applications and node groups. With an optimal scheduling strategy, you can decrease the cost of total resources allocated.

Optimize your autoscaling

Kubernetes is designed to support your applications with the **processing power and services they require on an as-you-need-it basis.** With autoscaling, Kubernetes can quickly adapt to the change in demand by ensuring that the right size and number of pods are being used. This results in improved performance while reducing resource wastage and cost. Yet, not all autoscaling setups are created equal.

Flex

Can you be flexible? Certain workloads are suitable for spot instances. Spot instances enable cloud providers to sell extra capacity; this means that their availability is closely coupled with demand and not guaranteed. Typically you will enter a bidding process in which you specify a price-per-hour you are willing to pay. Implementation is worth the effort, because spot VMs can significantly reduce your costs — if you want to learn more about when and how to apply spot instances, see our guide.

Configure quality of service

Kubernetes configurations provide three Quality of Service (QoS) classes for pods: Guaranteed, Burstable, and Best Effort. These classes help achieve **higher utilization of node resources**, resulting in less wastage and more cost benefits. Based on the defined resource limit and request parameters, you can assign pods to one of the three classes. QoS classes help the Kubernetes scheduler determine the scheduling of the pods on nodes — ensuring the nodes' resources are maximally utilized. They also decide the order of the eviction priority of the pods when a node gets low on resources.

What next?

As you may have noticed, real-world resource usage in the highly volatile K8s environment means that tracking actual usage levels and performing cloud cost management to distribute overhead expenses is no small challenge. Traditional approaches to calculating resource consumption and related costs are inefficient, propagate inaccuracies, and often are too perplexing to even achieve manually.

In this guide, we have presented several comprehensive and practical strategies that you can apply to mitigate these issues. Whether you are deploying Kubernetes clusters directly or with a cloud service provider such as AWS EKS, a robust K8s labeling strategy pays huge dividends. And, when it comes to cost savings, those six areas for optimization that we detailed can have a significant impact on the bottom line.

Identifying your K8s units are just the starting point to creating unit of economics to inform your FinOps teams. Applying the FinOps methodology helps with cloud financial management by blending cost, agility, and quality at the pod level.

We are, of course, not impartial observers of the cloud cost ecosystem here at Finout! With Finout, you can integrate with cloud providers and observability platforms to combine your business metrics with costs in order to reveal your unit of economics and make better-informed business decisions. Finout's Kubernetes cost monitoring tools empower you to precisely allocate your Kubernetes costs to your business unit. Want to see Finout in action? Book a demo today.