FinOps Foundation

FinOps for Kubernetes: Unpacking container cost allocation and optimization

www.finops.org

FinOps for Kubernetes: Unpacking container cost allocation and optimization

Addressing how and why containers complicate how teams manage the visibility and optimization of container costs, and how to apply FinOps best practices to improve cloud financial management.

Before you begin reading this paper

You should understand the basics of how cloud computing works, know the key services on your cloud providers, including their common use cases, and have a basic understanding of billing and pricing models. Being able to describe the basic value proposition of running in the cloud and understand the core concept of using a pay-as-you-go consumption model are also necessary.

You'll also need to have a base level of knowledge of at least one of the three main public cloud providers (AWS, Azure, Google Cloud). For AWS, we recommend AWS Business Professional training or, even better, the AWS Cloud Practitioner certification. For Google, check out the Google Cloud Platform Fundamentals course. For Azure, try the Azure Fundamentals learning path. Each can usually be completed in a full day workshop.

Introduction

Since containers and container orchestrators are becoming a popular choice for many teams, it's vital to understand the fundamental impact of these containerized workloads on FinOps practices. Shared resources, like containers, create challenges with cost allocation, cost visibility, and resource optimization.

In the containerized world, traditional FinOps cost allocation (e.g. mapping costs of resources back to teams or projects one-to-one) doesn't work. You can't simply allocate the cost of a resource to a tag or label, because resources may be running multiple containers, with each supporting a different application. They also may be attached to different cost centers around the organization.

Whether you're a part of a team with an established FinOps practice, or are building up the discipline, everyone can relate to the challenges of mapping cloud utilization cost to their drivers one-to-one. And then come containers and Kubernetes. All of a sudden, those foundational cloud finance management practices need to be tweaked a bit. Have no fear though-- the same FinOps principles and practices can help your teams track containerization spending accurately.

We created this whitepaper to cover key questions, challenges, and considerations that any FinOps teams should understand before tackling containerization in the cloud. If you're already using containerized services, then use this guide to double check that your teams are operating with strong, modern FinOps best practices.

This whitepaper will go over general container terms and concepts. It will also explore and address Kubernetes and container costs from the lens of FinOps core practices:

- Inform: Understand how the public cloud providers charge for their Kubernetes services. Learn why performing chargeback and cost allocation in environments with Kubernetes is challenging.

- Optimize: Look for ways to continuously optimize your Kubernetes clusters and pods.

- Operate: Determine your approach to container cost allocation and build policies to govern container spending, and integrate tooling to help manage container costs alongside traditional cloud spending.

Let's begin.

Looking at Kubernetes costs from a FinOps angle

When you look at the challenges that containerization poses for FinOps—cost visibility, cost showback/chargeback, and cost optimization—you quickly realize that you're encountering the same difficulties you faced as you moved into the cloud.

Containers represent another layer of virtualization on top of cloud virtualization, which creates a new wrinkle of complexity when it comes to tracking its costs. We'll be using fundamental FinOps practices to solve these challenges.

This paper will walk through container and Kubernetes FinOps challenges into the same **inform, optimize, and operate** lifecycle that you would apply to the broader cloud FinOps.

Relevant Terms and Concepts

Let's quickly run through the basics for anyone not familiar with containers or Kubernetes before we go further. It will also be helpful to create a common understanding of these components throughout this whitepaper.

Note: While there are many similarities between AWS Elastic Container Service (ECS) and Kubernetes, there are different terms used within each. For simplicity—besides when talking about Kubernetes specifically—we refer to "containers" and "server instances" where Kubernetes would refer to "pods" and "nodes."

Containers are, quite simply, a way to package software. All of the requirements and settings are baked into a deployable image. Container orchestration tools like Kubernetes help engineers deploy containers to servers in a manageable and maintainable way.

There are a few key terms we will use throughout this chapter:

Term	Definition
Image	A template of a container with the software that needs to be run.
Container	An instance of a container image; you can have multiple copies of the same image running at the same time.
Server instance/node	A cloud server (e.g., EC2 instance, virtual machine).
Pod	This is a Kubernetes concept. A pod consists of a group of containers and treats them as a single block of resources that can be scheduled and scaled on the cluster.
Container orchestration	An orchestrator manages the cluster of server instances, and also maintains the lifecycle of containers/pods. Part of the container orchestrator is the scheduler, which schedules a container/pod to run on a server instance. Examples include Kubernetes or AWS ECS.
Cluster	A group of server instances, managed by container orchestration.
Namespace	Another Kubernetes concept, a namespace is a virtual cluster where pods/containers can be deployed separately from other namespaces.
Pod labels	Pod labels are key/value pairs that can identify attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system. Each object can have a set of key/value labels defined. Each Key must be unique for a given object. These can be helpful when you want to group more than one namespace, for example.

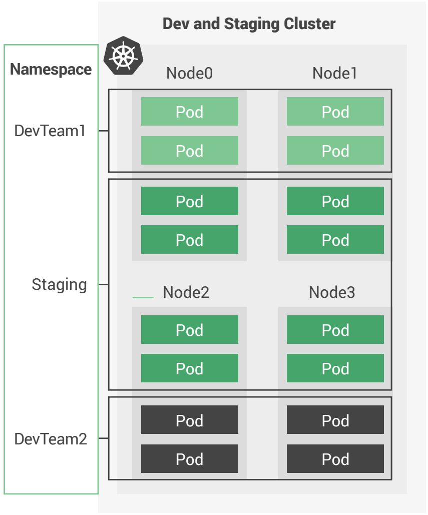

Figure 1: Basic view of a Kubernetes cluster

A Kubernetes cluster (see Figure at left) consists of a number of nodes (server instances) that run containers inside pods. Each pod can be made up of a varying number of containers. The nodes themselves support namespaces used to isolate groups of pods.

Inform: How container and Kubernetes costs are generated

Your first focus should be to generate reporting that determines the cost of individual containers on the clusters that teams are operating. This is a foundational step toward building visibility into your container spending.

How container cost allocation is different

Cloud service providers will charge your teams for every server instance that makes up a cluster. These charges are incurred when containers are deployed into a cluster, as they consume some amount of the cluster's resource capacity. The moment a process is run, a charge is incurred. It's similar to when you provision a server with your cloud service provider. You pay for that server resource, whether you use it or not.

In order to allocate the individual costs of a container that runs on a cluster, you'll need some way to determine how much of the underlying server the individual container consumed. You also need to understand the satellite costs of a running cluster. These include management nodes, data stores used to track cluster states, software licensing, backups, and potential disaster recovery costs. These costs are all part of running clusters and must be taken into account in your cost allocation strategy.

Container features bring up cost complexity

What users enjoy about container orchestration services is that they can greatly simplify:

- Scheduling how your teams spin up containers based on utilization requirements
- Managing network scaling and connection at various granularities
- Autoscaling and automating node creation and deletion to adapt to your changing workloads

You can run Kubernetes on virtual machine (VM) services provided by all major cloud providers, or use their native offerings to manage Kubernetes clusters. Examples include:

- Amazon Elastic Kubernetes Service (Amazon EKS) on AWS

- Google Kubernetes Engine (GKE) on Google Cloud
- Microsoft Azure: Azure Kubernetes Service (AKS)
- IBM Cloud: IBM Cloud Kubernetes Service
- Oracle Cloud Infrastructure: Oracle Container Engine for Kubernetes
- Alibaba Cloud: Container Service for Kubernetes (ACK)

Every provider will offer their own levels of support and service, including helping your teams migrate to the cloud. All of these services are like other cloud services: pay-as-you-go. Though pay-as-you-go cloud billing can be convenient, the ability of a cluster to run multiple projects from multiple teams can make cloud financial management and cost allocation a challenge.

The traditional approach to billing for managed kubernetes services is to charge per cluster per hour, plus the additional underlying resources that the cluster consumes. This makes container cost management especially challenging since you can't simply look at your cloud bill and see which resources are being consumed by a container cluster.

Why is it harder to bill and report on Kubernetes costs?

Allocating costs in a container environment surfaces additional challenges than traditional cloud environments. In a traditional cloud environment, FinOps practitioners can tag non-container resources one-to-one, allowing reporting tools to easily map services to cost centers. Assigning accountability for those apps is simply a mapping exercise of app vendor tags to a team.

With Kubernetes, one-to-one mappings of tags to teams don't cover some of the complex use cases that container utilization can create. Most Kubernetes clusters are shared services with applications run by any number of teams. This means there's no direct cost to a specific container; a series of them are coming together to generate costs per cluster of work. There is no easy way to map cloud charges to specific container usage.

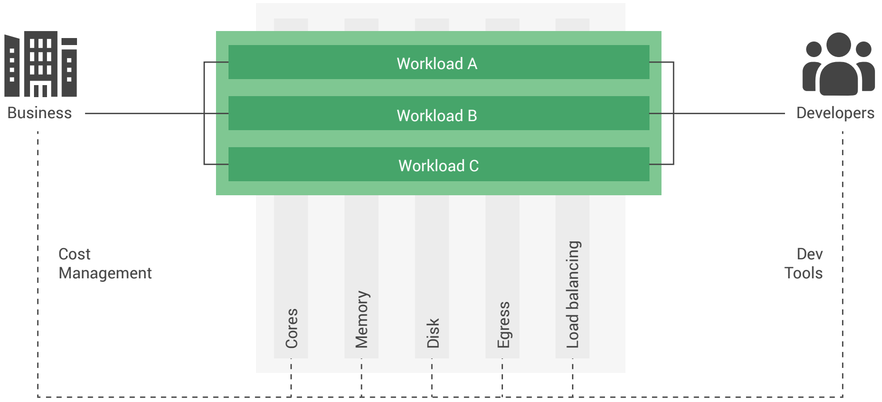

Figure 2: A high-level look at how containerized services can be labeled and mapped.

Source: Debo Aderibigbe, Billing Product Manager, Google Compute Cloud, FinOps Summit: Cost Visibility and Optimization in Kubernetes, KubeCon, CloudNativeCon EU 2020.

Containers are deployed in Kubernetes clusters, which consume cloud resources (such as compute) just as any other tenant would. The challenge lies in the fact that, within each cluster, you generally have multiple teams consuming portions of those underlying resources.

Additionally, containerized environments are much more dynamic than non-containerized ones, with the average lifespan of a container being one day versus a typically much longer utilization time for a VM. Given the dynamic nature of the Kubernetes scheduler, workloads can be rescheduled across instance type, zone, or even region. This makes cost management even more complex, as you must keep up with the rapid pace of change.

Tracking shared cluster and off-cluster resource costs together

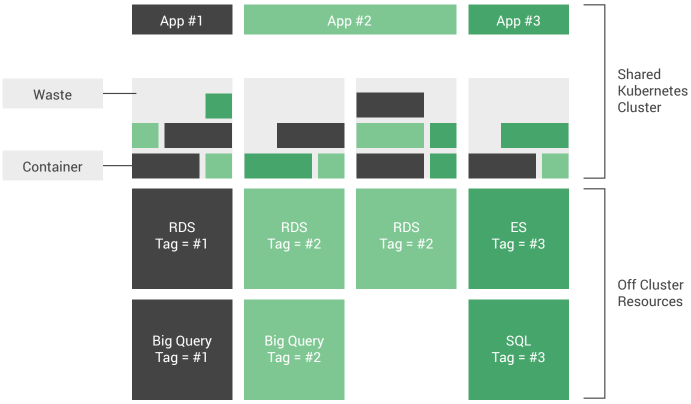

Figure 3: Applications with Shared Infrastructure and Services (Kubernetes)
Source: Casey Doran, “FinOps Summit: Cost Visibility and Optimization in Kubernetes”

The reality is that both models – your container costs from your shared clusters and from your cluster resources – need to be considered, tagged, and properly allocated. Teams using Kubernetes will also have non-Kubernetes resources, and need a cost model that can take both into account. Aligning your Tag strategy with your Kubernetes labeling strategy is critical for complete allocation.

So, how do we unravel this complexity and improve how we manage container costs? First, let’s quickly review how our teams are billed for Kubernetes usage.

Cost allocation practices and policy examples to govern container spending

With cloud financial management, predictability is king. Experienced FinOps practitioners will keep costs within budget and minimize surprises. Teams that are used to data center costs know that they are fixed and recurring. The transition to cloud services changes this expectation, and containerization further complicates things.

Container classes within Kubernetes

Cluster orchestration solutions like Kubernetes allow you to set different resource guarantees on the scheduled containers called Quality of Service (QoS) classes.

Guaranteed resource allocation

For critical service containers, you might use guaranteed resource allocation to ensure that a set amount of vCPU and memory is available to the pod at all times. You can think of guaranteed resource allocation as reserved capacity. The size and shape of the container do not change.

Burstable resource allocation

Spikey workloads can benefit from having access to more resources only when required, letting the pod use more resources than initially requested when the capacity is available on the underlying server instance. Burstable resource allocation is more like the burstable server instances offered by some cloud service providers (T-series from AWS and f1-series from GCP), which give you a base level of performance but allow the pod to burst when needed.

Best effort resource allocation

Additionally, development/test containers can use best-effort resource allocation, which allows the pod to run while there is excess capacity, but stops it when there isn't. This class is similar to preemptible VMs in GCP or spot instances in AWS.

When the container orchestrator allocates a mix of pods with different resource allocation guarantees onto each server instance, you get higher server instance utilization. You can allocate a base amount of resources to fixed resource pods, along with some burstable pods that may use up to the remainder of the server resources, and some best effort pods to use up any spare capacity that becomes available.

Now that we've covered somebasics, the next part of this section will start to explore some tactics to begin container cost allocation. We'll cover:

- Creating better understanding between dev, ops, and finance to work toward a more predictable budget. Using tools to automate away human error and use containerization services with more cloud financial accuracy.

- Using resource or application-defined aggregation: Stopping individual teams from creating many unique projects and instead, using defined and dedicated clusters.

- Assigning teams to namespaces can help provide accurate means to charge back container costs to respective projects and teams.

- Taking cloud service tagging to the next level with container cluster labels.

What are customers looking for more specifically?

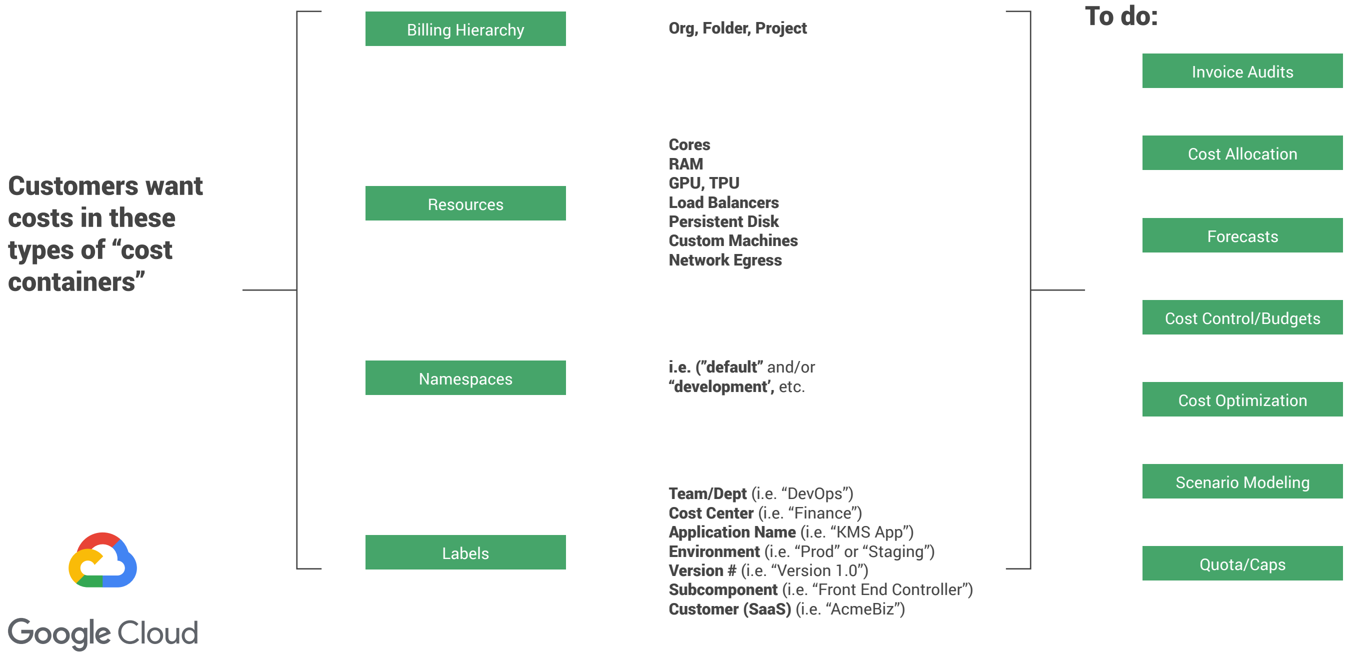
Customers want costs in these types of "cost containers"
Billing Hierarchy
Org, Folder, Project
Resources
Cores
RAM
GPU, TPU
Load Balancers
Persistent Disk
Custom Machines
Network Egress
Namespaces
i.e. ("default" and/or "development', etc.
Labels
Team/Dept (i.e. "DevOps")
Cost Center (i.e. "Finance")
Application Name (i.e. "KMS App")
Environment (i.e. "Prod" or "Staging")
Version # (i.e. "Version 1.0")
Subcomponent (i.e. "Front End Controller")
Customer (SaaS) (i.e. "AcmeBiz")
To do:
Invoice Audits
Cost Allocation
Forecasts
Cost Control/Budgets
Cost Optimization
Scenario Modeling
Quota/Caps
Google Cloud

Figure 4: A look at how complicated cost allocation for containers can be.
Source: Debo Aderibigbe, Billing Product Manager, Google Compute Cloud,
FinOps Summit: Cost Visibility and Optimization in Kubernetes, KubeCon, CloudNativeCon EU 2020.

In any FinOps or cloud financial management strategy, flexibly understanding costs can drive more accountability. The goal: to help users see their container cost drivers by both services used and container workload. Combining these two layers helps teams determine their Total Cost of Ownership.

Taking a deeper look at specific containerization costs

One method, recommended by Debo Aderibigbe, a Google Cloud Billing Product Manager, is to break down costs by:

- **Billing Hierarchy:** Organizations, folders, projects, normalizing them with cross-cloud concepts: Linked Accounts, Tags, kSubscriptions, etc.
- **Resources:** Compute cores, RAM, GPU, TPU, Load Balancers, Persistent Disk, Custom Machines, Network Egress
- **Namespaces:** labeling specific, isolated containers
- **Labels:** Teams, cost centers, app names, environment, and more

With a deep labeling and tagging of all of these cost drivers, users can improve the accuracy of how they invoice teams, audit costs, allocate costs, optimize overrun costs, model budgeting scenarios, or fit workload costs within quotas or under budget caps.

Consistent labeling and namespace strategy to improve allocation

Once you've implemented a consistent and robust labeling and namespace strategy, you can start to consider how you will allocate cluster costs. Unless you're using GKE, you can't easily see which groups are driving costs within a cluster.

A common methodology will be to look at the proportional resources consumed by each group (label, namespace, etc) and use that to allocate the cluster costs to those groups. For example, if you have four namespaces in a cluster that each consume 25% of the cluster resources, you could decide to allocate 25% of the total cluster costs back to each of those namespaces.

In the real world, any environment will never be this simple or straightforward. One additional layer of complexity is answering the question of how are you determining cluster resource utilization? Will you base it off CPU, memory, or a combination of the two? Do you want to consider requests or actual consumption?

There are pros and cons to each of these approaches, as outlined in the table below.

	Resource Requests	Actual Usage
Advantages	• Allocate all costs • Incentivize teams to only provision what they need • There are tools to help! (e.g. vertical pod autoscaler)	• Each team / app only pays for what they use
Challenges	• Some organizations are not using resource request fields yet • May also incentivize under-specifying requirements	• Who pays for the rest (idle time / cycles)? • What do we do about overprovisioning? • Can incentivize teams to provision more just in case, and not pay for it • Can set unrealistic goal of 100% utilization

Going beyond the Core Cluster Costs

When allocating the costs out to the consumers of the cluster, it's important to consider not only the cost of the compute nodes the container operated upon, but also the satellite costs of operating the cluster.

Management / Cluster Operational costs

Costs charged by the cloud service provider for managing the cluster or costs incurred by running self managed container orchestrator nodes should be considered. Edge services like WAF, Load Balancers, etc also contribute to the overall cost of running a workload on a cluster.

Storage Costs

Containers consume storage even if this is treated as ephemeral by the services running inside the container. Outside of the container however, consider the host OS on the nodes and any backup or data retrieval storage that is used in operating a production cluster can be allocated back to the workloads running on the cluster.

Licensing

Licensing costs are always a fun topic, if you are running licensed operating systems for the host node. License costs may be included in the charge by your cloud service provider. However, if you operate these using bring your own license (BOYL), the license cost will need to be allocated from the external spend. Alongside the host operating system, consider any software packages running on the host OS that incurs a license fee. The workload itself running inside the container may also be using licensed software that may need to be allocated.

Observability

Often, metrics and logs are sent from the cluster to a service which your teams are able to visualise, monitor, and alert upon. This data is sent either to services operated by the cloud service provider or even 3rd party SaaS solutions like (Splunk Cloud, Sumo Logic, Datadog, SignalFx, etc).

Security

The major cloud service providers now have very extensive security related services to assist in maintaining a secure cloud environment. Enabling these security features however does not often come for free, and these additional costs may need to be allocated to your teams.

Tempting as it may be to include every individual dollar from all of the above sources in your cost allocation strategy, as with everything FinOps, we recommend you start simple and grow your practice over time (Crawl, Walk, Run). It can become overwhelming to implement all of these cost allocation items at once, and as you develop both a process to allocate costs and the understanding of the allocation around your organization, the divide and conquer approach will be more likely to succeed.

Addressing static versus runtime container costs

Containerization costs are also broken up into two primary types: Static and Runtime costs.

Static costs

For static costs, you need to consider the creation of the solution within the container to ensure the quality of the solution to the project but also how it affects the CPU, Network and Storage when deployed. Static container costs can be further defined by stateless and stateful containers: Stateless examples of these include:

- Web servers with static resources: Apache, Nginx, IIS,
- application servers, stateless applications: Tomcat, nodeJS, JBoss, Symphony, .NET
- Microservices; Spring Boot, Play, Quarkus
- Tools: Maven, Gradle, scripts, tests

Application servers with stateful applications

There is often a need to store user sessions in an application. Two approaches to handling this case are to use a load balancer with session affinity to ensure the user always goes to the same container instance or to use an external session persistence mechanism which all container instances share.

There are also some components that provide native clustering such as portals or persistence layer caches. It's usually best to let the native software manage synchronization and states between instances. Having the instances on the same overlay network allows them to communicate with each other in a fast, secure way.

Databases

Databases usually need to persist data on a filesystem. The best practice is to only containerize the database engine while keeping its data on the container host itself. This can be done using a host volume, for example: $ docker run -dit -v /var/myapp/data:/var/lib/postgresql/data postgres.

Kubernetes can also be used as an alternative to managed database services. For example, a cluster dedicated to MongoDB or Elasticsearch can deliver something similar to a fully-managed service for a fraction of the cost.

Applications with shared file systems

Content Management Systems (CMS) use filesystems to store documents such as PDFs, pictures, Word files, etc. This can also be done using a host volume which is often mounted to a shared filesystem so several instances of the CMS can access the files simultaneously.

Runtime costs

Runtime costs by most are assumed to be static for containers. How you run your containers will affect your bottom line. Examples of these costs from cloud service providers include:

- Bandwidth is often overlooked or underappreciated in estimating cloud computing charges.
- Leaving a containerized application deployed that you forgot about is a surefire way to get a surprising bill. Once you put applications or data into the cloud, they continue to cost you money, month after month, until such time as you remove them. It's very easy to put something in the cloud and forget about it
- Compute charges are not based on usage.
- Polling data in the cloud is a costly activity and incurs transaction fees. Very soon the costs could add up based on the quantity of polling.
- Unintended traffic in the form of DOS attacks or spiders etc. could increase traffic in unexpected ways. The best way to deal with such unintended charges is to audit the security of the application and provide measures of controls such as CAPTCHAs.
- Management: Regularly monitor the health of your applications and its billing. Regularly review whether what's in the cloud still needs to be in the cloud. Regularly monitor the amount of load on your applications. Adjust the size of your deployments to match load.

How LiveRamp addressed containerization costs on GKE

The LiveRamp team migrated on-prem services to Google Cloud Platform, scaling up container services (and costs), which caught the eye of the Finance team. They completed a successful technical migration, but a month later, budgets were overrun. They needed a way to explain what happened.

At that time, LiveRamp was missing the cultural shift to FinOps, and developers didn't understand their new responsibilities were to cloud finances. To increase this understanding, container-focused FinOps principles helped rationalize cloud cost and operational decisions in different ways. They had to overcome challenges that came up due to every team having their own accounts and setups.

After much FinOps-focused work, new policies were built in, such as enforced name-spacing for large clusters and better tagging and labeling to improve accuracy of allocation, both key FinOps practices. It also helps to find tooling that suits your teams needs in managing cloud financial management, e.g. Cloudability, CloudHealth, native tools, etc.

-Sasha Kipervarg, (former) Head of Global Cloud Operations, LiveRamp

Considerations for container savings in production

Containerized deployments can realize up to 90% in discounts compared to on-demand prices for running stateless and fault-tolerant applications. Containers that shall be ephemeral and stateless adhere to a graceful startup and graceful shutdown.

With these qualities, serverless deployments become more attractive due to the fact that these deployments are only charged when running. Another way to think about it is that you're charged nothing while in a dormant state. Just deploying the contents to a serverless API is not enough as you must obtain equal functionality with performance. The cold start becomes your nemesis.

However, new boundaries are opening up that allow universal batch serverless loads to run on cloud providers, such as Apache Beam. Once enabled, many new areas from IT will be able to participate in the savings, following sophisticated data-parallel processing pipelines that enable execution across a diversity of cloud provider engines, or runners.

Optimize: Build in cost efficiencies for your Kubernetes environment

Like the inform phase, the optimize phase in the container world is a direct adaptation of your original FinOps practices built for cloud platforms, but applied with the complexities of containers in mind.

One can argue that containerization solves the rightsizing problem. Having more workloads running on the same server instance appears more cost-effective. But as you've seen so far, it's a complex task to measure which teams or projects generate these costs, or whether or not you're getting savings from your clusters.

Let's take a look at how your FinOps practices have to evolve in order to be successful.

Once you've successfully charged back your kubernetes costs, the next step is to look for ways to optimize your kubernetes environments to reduce costs. There are several steps FinOps practitioners can partner with DevOps teams on to ensure that kubernetes costs don't accelerate out of control.

Pod / Container Rightsizing

Ensure that your containers are asking for an appropriate amount of resources. Since asking for too little means your application is not able to perform, often there is some buffer between the **requests** or **limits** configured for a container and what it really needs.

When this buffer is larger than necessary is when there is opportunity for cost savings. The Vertical Pod Autoscaler (VPA) is an example of an open source project that will help you by automatically adjusting the **requests** and **limits** configuration based on how much a container is seen to use, thereby saving you resources and cost while reducing overhead.

The Horizontal Pod Autoscaler (HPA) is meant to scale out and in rather than up and down for the workloads. Caution here is to make sure the VPA and HPA policies don't interfere with each other.

Binning and packing density settings are important and should be reviewed when designing clusters for purpose or business class tier of service where development or cost-focused clusters are meant to get as many containers per host. Meanwhile costs where production or performance clusters settings would be scoped for different needs and patterns. Having the ability to match the right quantity of pods and namespaces per instance family for your app is a good way to build a reference model that ensures proper capacity, availability, overhead, and economics.

Another thing to look out for is making sure Ingress controller settings for ensuring proper traffic shaping and load mgmt to containers are being leveraged.

Node Rightsizing

Next is the choice of worker node type for the cluster. This becomes a type of bin packing problem except with all the complexities of the various platform choices.

Often simply making incremental improvements, for example when you notice that your nodes always have excess memory, it can make sense to switch to a node type that is the same but offers slightly less memory. In practice however this can be more complex. For example, if you have workloads that often consume more than they **requested** and distinguish the difference between those cases where it was needed and where it was only consumed because it was available but wasn't critically needed.

Consider using instance weightings scores while you are producing a whitelist of instance size/types that are a good fit to run. The instance weighting will be useful when you are relying on diversified allocation strategies in a spot market where pricing may be the same but will help ensure the right value for your code is going to be provisioned based on weight values.

Autoscaling Adjustments

Something that makes Kubernetes especially powerful is the wealth of autoscaling options and the ability to respond dynamically to different conditions, such as increased or decreased demand. This can take some architecting and iterative adjustments to get right for your application, and there is room for waste along the way. However, the more tightly your horizontal pod autoscaling (when we need more / less pods) and cluster autoscaling (when do we need more / less nodes) are configured, the less waste and unnecessary cost to run your application.

Unraveling complexities from large-scale, multi-cloud enterprise container utilization

Just when you think you have containerization and cluster costs somewhat under control, a merger and/or acquisition comes along and adds another cloud service provider or two. This can complicate how the company conducts complete chargebacks and cost allocation.

Having a cloud cost management platform, whether native tooling or a dedicated application, can reduce the complexity and friction of allocating container costs. Cloud cost management platforms are most effective when the fundamentals are covered:

- Observing utilization by Kubernetes constructs and associating it with cloud billing data
- Use Label Key/Value pairs in Kubernetes to align with internal cost centers
- Unifying Kubernetes label keys with traditional key tags to deepen the allocation model

With these fundamentals in place, FinOps teams can account for Kubernetes cost in a holistic way, no matter how complex an enterprise's infrastructure might be. They can analyze costs by cluster or namespace, by cost center, and fully and accurately allocate these costs correctly.

Casey Doran,
Director of Product,
Apptio Cloudability

Discount types

Most cloud environments offer discount options that can offer significant savings, so long as the terms work for you and your application.

Spot / Low Priority / Preemptible worker nodes

These go by different names but typically provide a discount for workloads that may disappear on short notice. This allows cloud providers a way to incentivize filling all capacity pockets while being able to adjust for incoming workloads of priority that will pay on demand rates.

The declarative nature of Kubernetes makes it more conducive to taking advantage of spot discounts, as interruptions in the worker node fleet will be noticed and replaced. Still, some applications may not be tolerant of these interruptions. Applications that may fit this profile include data intensive workloads that may run in batches or are not as time sensitive such as machine learning training.

Commitment discounts

These also go by several names and mechanisms such as Reserved Instances, Savings Plans, Committed Use Discounts, Subscription Discounts, etc., but all typically offer a discount to users in exchange for a long term commitment to spend with that cloud provider. These commonly cover compute resources, and for users who expect to have consistent usage for a year or more on a given cloud (and in some cases for a given workload type), this is an option to lower your cluster costs.

The ability to enable a purchase method mix and declare the split percentage of on demand, committed, and spot variable pricing is a great way to automate that into a cost efficiency blueprint.

For example: 10% committed use for control plane, 90% spot for workers/replicas for non prod, 50/50 for prod etc.

Producing the pre-built helm charts or launch plans ahead of time and publishing for reusability, with all the various optimizations included, ensures the wisdom can be put into practice with minimal experience and friction.

“Kubeless” or “Serverless Containers” are an emerging option where the cloud provider is providing a higher order service that is consumed and may solve skills shortages by the customer in operating and administering Kubernetes clusters. Others have looked to Kubeless solutions on FPGA type instances where results are dramatically faster, much lower cost, and without the operating system (OS) and container drag along of what is needed to log, patch, monitor, and maintain an OS and Kubernetes cluster.

Operate: Build policies and practices to manage Kubernetes costs

Now that Kubernetes costs can be accurately allocated, and ways to optimize those costs are known, it's time to build a sustainable practice to enforce FinOps-focused policies and exercises. For example, scheduling development containers to be turned on and off around business hours, finding and removing idle containers, and maintaining container labels/ tags by enforcement are just some of the practices to teach and empower teams to utilize.

Tooling option to manage container costs

Most companies on the path toward mastering FinOps for cloud services, including container costs, often take one of three paths. Whether it's building in-house tooling to DIY their way to success, using native tooling provided by a cloud service provider, or using a cloud financial management platform, every choice has its pros and cons.

DIY tracking of container costs

If your teams have the talent and resources to do so, creating a DIY approach to tracking cloud financial data, including container costs, can be a way to make sense of this spending. Companies at massive scale are doing this today, including

Start your FinOps team with a strong foundation

Before adding practices and policies to better track containerization costs, ensure that your teams have strong cloud financial management fundamentals. Work with your finance team and overall center of cloud cost of excellence to get aligned and have strong answers to the following questions:

- Do you have established reporting capabilities and KPIs across the infrastructure?
- Do your teams consistently run checks to ensure cloud financial governance and policies are being followed (understanding how to use namespaces, tagging untagged resources, using semantic tags to relate to projects/teams, automating tag correction to reduce errors)?
- How well do your teams provision only what they need? Can they explain what those costs are based on and report on them accurately?
- Who pays for shared, common services?
- Do your teams actively and frequently identify opportunities to optimize cloud resource utilization (and are they empowered to do so?)?

Spotify, who run their own set of cloud management tools via backstage.io.

This also might make sense if your teams are in early stages of cloud utilization and generate billing data that can be managed this way.

Using native tooling to track costs

Using native tooling, often in conjunction with internal tools, can be another path toward FinOps success. As long as there's a means to access and digest cloud cost data to generate chargeback and reporting that makes sense to your business, you will likely be in good shape. Unfortunately, if your teams are running a multi-cloud infrastructure, you'll likely have to use each respective tool provided by each platform.

You might also need another data visualization or analysis tool to aggregate all of this cross-cloud financial data into one view.

Using a cloud financial management platform

When you get to a certain scale, manual efforts can cause more pain than good and it's time to use native tooling or a dedicated cloud financial management platform. This takes away manual human error and leaves allocation work (assuming everything is tagged and named accurately) to the tooling.

This can improve visibility into shared Kubernetes clusters and their costs. It can also improve how costs are allocated by teams or projects. Cloud financial management platforms often have features that support multi-cloud billing data as well.

However, a large part of your FinOps practice will be locked into the feature sets and pricing provided by the third-party tool. You might also have to annually make business sense of pricing, contracts, and negotiations to get a fair deal that makes sense for the size of your cloud infrastructure. A good starting place is to look at the FinOps Certified Platforms at the FinOps Foundation.

Answering these questions not only requires implementing key FinOps best practices, so catch up on those first. They might also cloud cost management solutions to help teams see the same utilization and cost reporting and act together to increase efficiency.

Questions courtesy of
Jonathan Morin,
Sr. Product Manager,
CloudHealth by VMware

> **"Piping in cost as a first-class citizen alongside this data is an advanced FinOps move that lets engineers easily measure the impact of infrastructure and application-level changes."**

Webb Brown
CEO, Kubecost

Empower and incentivize developers to track their Kubernetes utilization

With a well tagged and labeled infrastructure, this should provide the foundation for Dev and FinOps teammates to build their own custom reporting to track utilization data. Whether it’s assisted via API from native tools or cloud financial management tools, empowering teams to create and manage their own monitoring can help quite a bit.

Most likely, your developers are already tracking key infrastructure metrics that help people monitor service uptime and performance in real-time. Augmenting these existing metrics with cost data can enable them to incorporate this information in their decision making process without major workflow changes.

Beyond costs that track spending and utilization, empower developers to further make their Kubernetes setups even more efficient by leveraging tools that are already part of their existing workflows. Oftentimes, this means embracing container-focused open source tools. Application management tools like Helm, monitoring solutions like Prometheus, container workflow engines like Argo, and service mesh solutions like Linkerd can really help engineer teams get actionable visibility for their Kubernetes clusters in near real-time.

Overcome Kubernetes cost management challenges with FinOps best practices

FinOps practitioners know that the cloud only gets more complicated from here, and that having strong cloud financial management practices eases how we report on and allocate costs.

This whitepaper outlined some strong foundational steps to start increasing the accuracy of how your teams report on container costs. Within this paper, we covered:

- **Inform:** How the public cloud providers charge for their Kubernetes services. We reviewed why performing chargeback and cost allocation in environments with Kubernetes is challenging.
- **Optimize:** How to continuously look for ways to optimize your Kubernetes clusters and pods, and recommended a few tactics to test.
- **Operate:** We outlined a few approaches to container cost allocation and build policies to govern container spending, and how to integrate tooling to help manage container costs alongside traditional cloud spending. We also reviewed different types of tooling that can help your teams build cloud finance policies, tracking, and governance.

As more users adopt and put containers and Kubernetes to work, new best practices form around tracking both utilization and finance data. This means our Finops lens on containers is constantly changing and evolving.

Join our community

Get a deeper dive into unwinding container cost and utilization data by joining the FinOps Foundation community. Members are constantly discussing container and Kubernetes best practices and sharing new tips and tricks. The #finops-on-kubernetes Slack room is a great place to join the conversation, you'll get a link to join once your membership is approved.

Check back and contribute your expertise

We invite any FinOps practitioner to check back for updates and contribute to the conversation by either joining our Community or checking out our various GitHub repos to lend your expertise.

Take a look at other Cloud Native Computing Foundation content

FinOps covers more of the operational and financial side of the cloud. The CNCF has a vast collection of resources to track other cloud-native projects. Check out their library of resources as well.

Acknowledgements

The FinOps Foundation extends its thanks to its community members and guest contributors for all their help with this project. The more our community gets together to tackle cloud cost optimization, the more we all collectively learn.

We'd like to highlight the following people for going above and beyond on this whitepaper: Debo Aderibigbe of Google Cloud Platform, Casey Doran of Apptio Cloudability, Jonathan Morin of Cloudhealth by VMWare, Mike Giacommetti, and Rachel Dines of CloudHealth by VMware, Sasha Kipervarg of Ternary, Webb Brown of KubeCost, J.R. Storment, Mike Fuller, Peter Treese, and Matt Leonard of Google Cloud Platform And to Chris Aniszczyk for suggesting the collaboration with CNCF!

FinOps Foundation

www.finops.org