Densify

Whitepaper

Capacity Operations

Continuous Compute Optimization for Cloud & Container Environments

Andrew Hillier
Co-Founder & CTO, Densify

What is Capacity Operations?

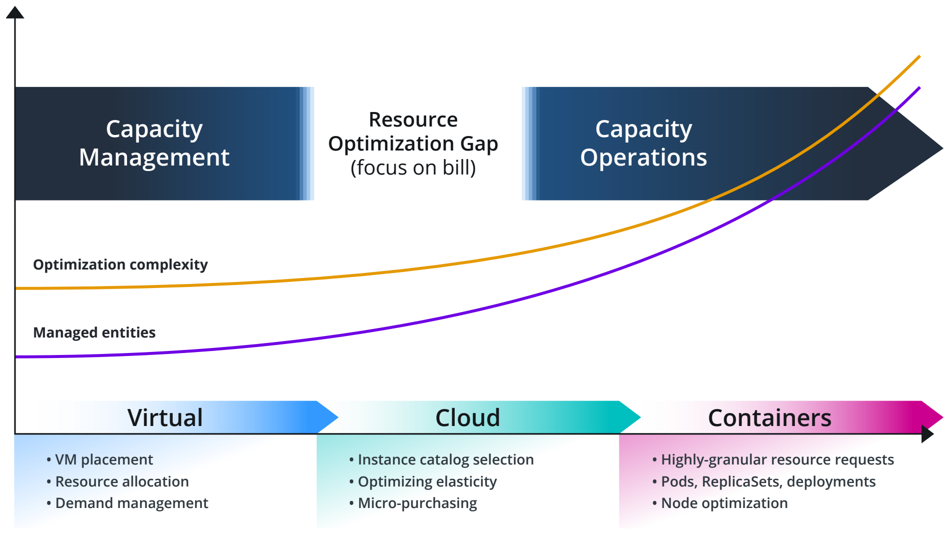
Capacity Operations, or "CapOps," is the emerging discipline of continuously optimizing compute resources in cloud and container environments. It fills a gap that has emerged between the DevOps and FinOps processes, where in-depth analysis of the ongoing resource requirements of cloud and container-based applications typically isn't performed by either group, leading to inflated bills and unnecessary operational risk. And although it is driven by the same general goals, CapOps differs from traditional capacity management in that the focus is less on long term planning to make sure there is enough "on the floor," and more on continuous alignment of application demands and infrastructure supply in elastic, "as-a-service" environments.

Background

Since the dawn of computing, there has been a need to ensure that IT environments have sufficient resources to meet application demand, without having too much. As the industry progressed from mainframes, to midrange, to open systems, to virtual environments, the practice of managing capacity evolved with it, ultimately resulting in a highly-mature discipline designed to minimize the risk of running out of resources, while at the same time ensuring that over-purchasing is avoided. Specialized activities such as demand management, risk management, predictive forecasting and others all helped contribute to the smooth and efficient operation of IT environments.

The advent of cloud computing created a disruption in many areas of IT, and the management of capacity was no exception. The ability to purchase resources "on demand" eliminated the need for long-term planning of hardware purchases, and also greatly reduced the risk of running out of resources. This caused the pendulum to swing away from capacity management and toward the bill, causing many capacity teams to be disintermediated in the process. The newfound ability to see costs broken down in extreme detail gave rise to a new focus, and a new breed of tooling, designed to understand, allocate, and minimize costs.

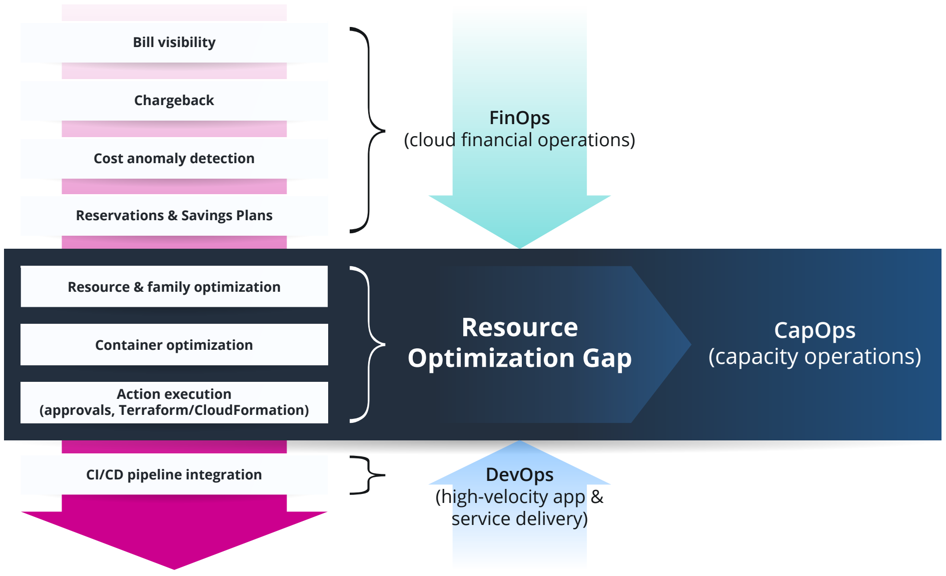
But, focusing on allocating costs and purchasing discounts to minimize the bill will only get you so far, and in many cases a high cloud bill is just a symptom of a deeper underlying resource problem. If applications are configured to use the wrong resources, and if the elastic structures in the cloud are not working efficiently, then no amount of discounting will claw back the extra cost. To truly optimize the efficiency of these environments, while at the same time ensuring performance requirements are met, the pendulum needs to swing back, and a more disciplined approach to optimizing the resources in use needs to be taken.



The shift to public cloud has created a blind spot for organizations where the actual resources being consumed are not being optimized—inflating bills and creating operational risk

Enter Capacity Operations

The logical path for this to take mirrors what has happened in the areas of development and financial management. Application development has become far more agile, and has effectively merged with certain aspects of operations to become DevOps. This mashup of an offline activity (development) and an online practice (operations) helped evolve application delivery into a much more agile, elastic and collaborative process. The same shift is also happening to financial optimization, where the offline practice of financial management is becoming more operational, producing a much more dynamic FinOps practice that is capable of keeping up with dynamic cloud and container environments.


Bill visibility
Chargeback
Cost anomaly detection
Reservations & Savings Plans
FinOps
(cloud financial operations)
Resource & family optimization
Container optimization
Action execution
(approvals, Terraform/CloudFormation)
Resource Optimization Gap
CapOps
(capacity operations)
CI/CD pipeline integration
DevOps
(high-velocity app & service delivery)

But both of these disciplines have practical limitations. DevOps has the mandate to "deliver applications and services at high velocity," but typically doesn't include detailed analysis and optimization of the resources used by those applications, either when they are initially deployed, or after they have been running. These teams are too busy focusing on new features and time-to-market, as they should be, since they are uniquely able to control this.

Similarly, FinOps has the mandate of "cloud financial operations," and is the formalization of the various financial practices surrounding cloud. And although optimizing resources has a significant impact on the financial picture, FinOps teams typically do not have the tooling, subject matter expertise, or bandwidth to delve deeply into detailed resource utilization, optimizing elasticity, sizing containers, or other highly-granular activities.

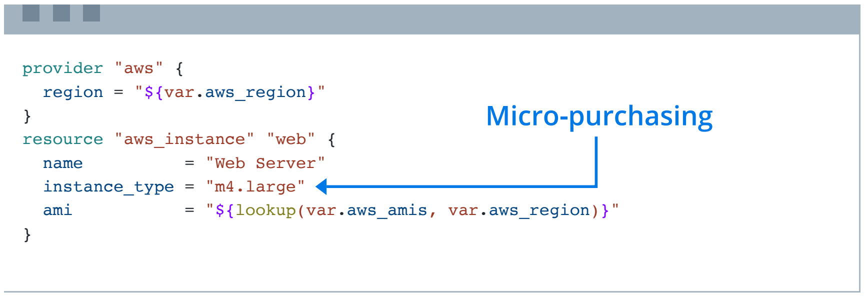
Following this pattern, the logical evolution of capacity is for it to transition from an offline practice (planning, management) to an online, more operational discipline. The resulting "CapOps" practice can be considered to have the mandate of "continuous resource optimization," and by refocusing on the new, more dynamic capabilities of cloud and container infrastructure, it can bring back the discipline that was temporarily lost. This allows organizations to once again ensure that there are "sufficient resources to meet application demand, without having too much," filling the gap left by the evolution of the DevOps and FinOps practices.

Key Capabilities of Capacity Operations

To understand the requirements of CapOps it is useful to draw parallels to the on-prem data center hosting model, and in particular, what it means to have infrastructure "on the floor" (and how that infrastructure gets on the floor). In an on-prem, CapEx-oriented hosting model there is a long lead time for deploying new compute resources, and this drives a lot of the capacity analysis that is performed, including forecasting and demand management.

Accurately modeling the pipeline of inbound demand, and ensuring resources are available to meet demand, can prevent unnecessary risk, and making sure those resources are used efficiently can prevent costly purchases.

Cloud infrastructure, on the other hand, enables you to deploy resources “on the floor” in minutes, or even seconds, through API calls or lines of code (such as Terraform or CloudFormation). This highly-elastic “micro-purchasing” model is a key advantage of the cloud, and eliminates the need for many of the planning-oriented capacity management activities.



```
provider "aws" {
  region = "${var.aws_region}"
}
resource "aws_instance" "web" {
  name          = "Web Server"
  instance_type = "m4.large"
  ami           = "${lookup(var.aws_amis, var.aws_region)}"
}
```

Automation APIs & Infrastructure as Code

HashiCorp Terraform

AWS CloudFormation

Red Hat Ansible

But, this doesn’t mean capacity can be ignored, as many organizations initially assumed, but rather it needs a completely different set of activities in order to optimize resources. In many cases these activities must be re-thought from the ground up, since the fundamental assumptions of traditional processes have changed. For example, even taking inventory of what is “on the floor” is very different than in on-prem environments, and now resembles more of a “stock chart” of ups and downs than a static number of things that can be counted. This fluidity has a ripple effect through many other areas, including capacity.

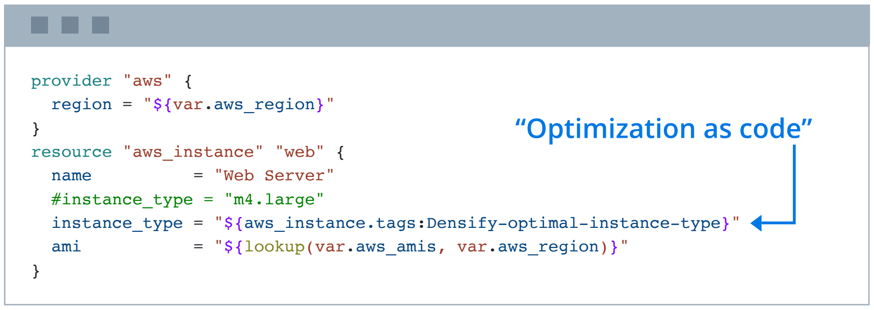
And this micro-purchasing model is a double-edged sword. While providing agility, it also puts resourcing decisions in the hands of engineers and developers who may not have sufficient information to make the right choice. In this new world, a relatively junior engineer can put a line of code in a file that causes a purchase, and although this purchase is small, getting it wrong across many instances can result in tremendous inefficiency and significant cost. As a result, even traditional capacity management activities such as rightsizing virtual machines now need to be done in a completely different way, and must adapt to this new form of decentralization of decision-making.

Given all of this, there are a set of fundamental operations that must be performed in order to make sure that the right resources are deployed at any point in time. For cloud environments, this includes:

- **Instance sizing** (upsize, downsize) and termination
- **Instance family optimization** (memory optimized, CPU optimized, burstable)
- **Scaling group node optimization** (node type, size)
- **Scaling group scaling parameter optimization** (elasticity)
- **DB-as-a-service optimization**

Only one of these operations, instance sizing, resembles something that is done in traditional virtual environments, but even this must be done very differently. As mentioned above, the resources in use are now typically specified in manifests, or "infrastructure as code," and any optimization must be embedded in these manifests, with automation occurring through the deployment pipeline. This is very different from virtual environments, where automation typically involves modifying the VMs directly—this approach will not work in environments that leverage infrastructure as code, as the running instances will always revert back to what the code says.

Instead, the optimization recommendations must also become lines of code to enable continuous optimization, effectively creating "optimization as code."



```
provider "aws" {
  region = "${var.aws_region}"
}
resource "aws_instance" "web" {
  name          = "Web Server"
  #instance_type = "m4.large"
  instance_type = "${aws_instance.tags:Densify-optimal-instance-type}"
  ami           = "${lookup(var.aws_amis, var.aws_region)}"
}
```

Automation APIs & Infrastructure as Code

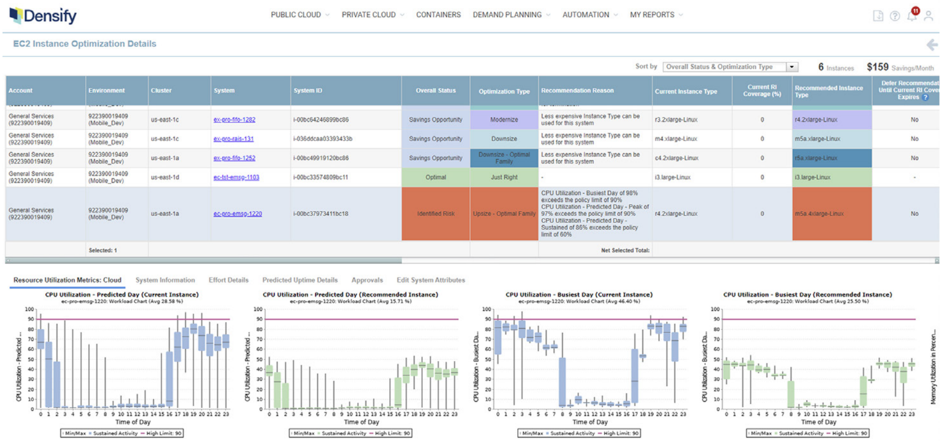
HashiCorp Terraform

AWS CloudFormation

Red Hat Ansible

Beyond the instance sizing, the rest of the operations are new. Clouds use catalog-based sizing, and optimization analysis must not only determine the correct instance size, but also the optimal instance family for a given workload, which can be complex to determine. Even within a given family, there may be newer instance types available that are faster, cheaper, or both, and modernizing to these new instance types can be a quick way to gain efficiency.

Building on this, the optimization of scaling groups also benefits from instance-level optimization, as it is common for there to be a mismatch between the resources being consumed by the applications and those being provisioned in the scaling groups. And scaling groups also enable the optimization of the scaling parameters in order to ensure that they are scaling up when needed, and down when not. Optimizing these settings enables organization to configure the cloud infrastructure to dynamically respond to load in an optimal manner, something that is not possible in legacy environments. This scaling group optimization is becoming increasingly important as organizations move to containers—container clusters typically run on auto scaling groups, and not only is container performance highly-dependent on them scaling properly, but the costs of the container environment are also reflected in the scaling group costs.

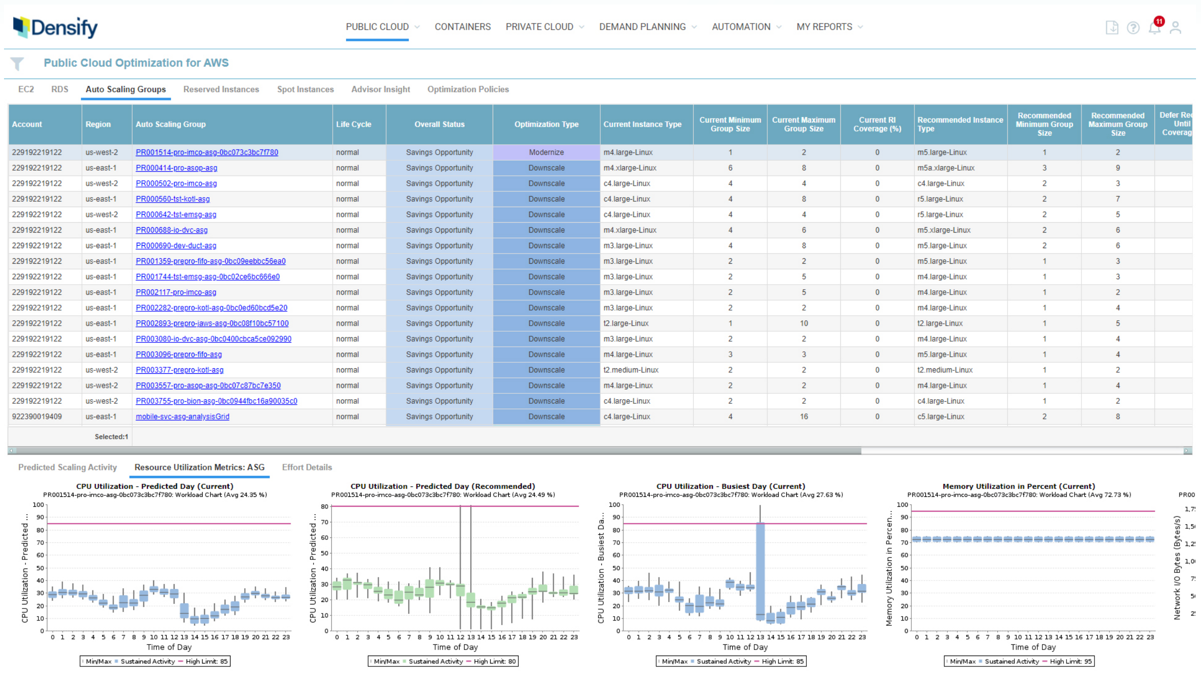


EC2 Instance Optimization Details

Sort by Overall Status & Optimization Type — 6 Instances — $159 Savings/Month

Account	Environment	Cluster	System	System ID	Overall Status	Optimization Type	Recommendation Reason	Current Instance Type	Current RI Coverage (%)	Recommended Instance Type	Defer Recommendat Until Current RI Cove Expires
General Services (922390019409)	922390019409 (Mobile_Dev)	us-east-1c	ex-pro-fifo-1282	i-00bc64246899bc86	Savings Opportunity	Modernize	Less expensive Instance Type can be used for this system	r3.2xlarge-Linux	0	r4.2xlarge-Linux	No
General Services (922390019409)	922390019409 (Mobile_Dev)	us-east-1c	ex-pro-rais-131	i-036ddcaa03393433b	Savings Opportunity	Downsize	Less expensive Instance Type can be used for this system	m4.xlarge-Linux	0	m5a.xlarge-Linux	No
General Services (922390019409)	922390019409 (Mobile_Dev)	us-east-1a	ex-pro-fifo-1252	i-00bc49919120bc86	Savings Opportunity	Downsize - Optimal Family	Less expensive Instance Type can be used for this system	c4.2xlarge-Linux	0	r5a.xlarge-Linux	No
General Services (922390019409)	922390019409 (Mobile_Dev)	us-east-1d	ec-tst-emsg-1103	i-00bc33574809bc11	Optimal	Just Right	-	i3.large-Linux	0	i3.large-Linux	-
General Services (922390019409)	922390019409 (Mobile_Dev)	us-east-1a	ec-pro-emsg-1220	i-00bc37973411bc18	Identified Risk	Upsize - Optimal Family	CPU Utilization - Busiest Day of 98% exceeds the policy limit of 90% CPU Utilization - Predicted Day - Peak of 97% exceeds the policy limit of 90% CPU Utilization - Predicted Day - Sustained of 86% exceeds the policy limit of 60%	r4.2xlarge-Linux	0	m5a.4xlarge-Linux	No
	Selected: 1						Net Selected Total:				

Resource Utilization Metrics: Cloud
System Information
Effort Details
Predicted Uptime Details
Approvals
Edit System Attributes
CPU Utilization - Predicted Day (Current Instance)
ec-pro-emsg-1220: Workload Chart (Avg 28.58 %)
CPU Utilization - Predicted Day (Recommended Instance)
ec-pro-emsg-1220: Workload Chart (Avg 15.71 %)
CPU Utilization - Busiest Day (Current Instance)
ec-pro-emsg-1220: Workload Chart (Avg 46.40 %)
CPU Utilization - Busiest Day (Recommended Instance)
ec-pro-emsg-1220: Workload Chart (Avg 25.50 %)
Time of Day
Min/Max Sustained Activity High Limit: 90



PUBLIC CLOUD CONTAINERS PRIVATE CLOUD DEMAND PLANNING AUTOMATION MY REPORTS

Public Cloud Optimization for AWS

EC2 RDS Auto Scaling Groups Reserved Instances Spot Instances Advisor Insight Optimization Policies

Account	Region	Auto Scaling Group	Life Cycle	Overall Status	Optimization Type	Current Instance Type	Current Minimum Group Size	Current Maximum Group Size	Current RI Coverage (%)	Recommended Instance Type	Recommended Minimum Group Size	Recommended Maximum Group Size	Defer Re Until Coverag
229192219122	us-west-2	PR001514-pro-imco-asg-0bc073c3bc7f780	normal	Savings Opportunity	Modernize	m4.large-Linux	1	2	0	m5.large-Linux	1	2	
229192219122	us-east-1	PR000414-pro-asop-asg	normal	Savings Opportunity	Downscale	m4.xlarge-Linux	6	8	0	m5a.xlarge-Linux	3	9	
229192219122	us-west-2	PR000502-pro-imco-asg	normal	Savings Opportunity	Downscale	c4.large-Linux	4	4	0	c4.large-Linux	2	3	
229192219122	us-east-1	PR000560-tst-kotl-asg	normal	Savings Opportunity	Downscale	c4.large-Linux	4	8	0	r5.large-Linux	2	7	
229192219122	us-west-2	PR000642-tst-emsg-asg	normal	Savings Opportunity	Downscale	c4.large-Linux	4	4	0	r5.large-Linux	2	5	
229192219122	us-east-1	PR000688-io-dvc-asg	normal	Savings Opportunity	Downscale	m4.xlarge-Linux	4	6	0	m5.xlarge-Linux	2	6	
229192219122	us-east-1	PR000690-dev-duct-asg	normal	Savings Opportunity	Downscale	m3.large-Linux	4	8	0	m5.large-Linux	2	6	
229192219122	us-east-1	PR001359-prepro-fifo-asg-0bc09eebbc56ea0	normal	Savings Opportunity	Downscale	m3.large-Linux	2	2	0	m5.large-Linux	1	3	
229192219122	us-east-1	PR001744-tst-emsg-asg-0bc02ce6bc666e0	normal	Savings Opportunity	Downscale	m3.large-Linux	2	5	0	m4.large-Linux	1	3	
229192219122	us-east-1	PR002117-pro-imco-asg	normal	Savings Opportunity	Downscale	m3.large-Linux	2	5	0	m4.large-Linux	1	2	
229192219122	us-east-1	PR002282-prepro-kotl-asg-0bc0ed60bcd5e20	normal	Savings Opportunity	Downscale	m3.large-Linux	2	2	0	m4.large-Linux	1	4	
229192219122	us-east-1	PR002893-prepro-jaws-asg-0bc08f10bc57100	normal	Savings Opportunity	Downscale	t2.large-Linux	1	10	0	t2.large-Linux	1	5	
229192219122	us-east-1	PR003080-io-dvc-asg-0bc0400cbca5ce092990	normal	Savings Opportunity	Downscale	m3.large-Linux	2	2	0	m4.large-Linux	1	4	
229192219122	us-east-1	PR003096-prepro-fifo-asg	normal	Savings Opportunity	Downscale	m4.large-Linux	3	3	0	m5.large-Linux	1	4	
229192219122	us-west-2	PR003377-prepro-kotl-asg	normal	Savings Opportunity	Downscale	t2.medium-Linux	2	2	0	t2.medium-Linux	1	2	
229192219122	us-west-2	PR003557-pro-asop-asg-0bc07c87bc7e350	normal	Savings Opportunity	Downscale	m4.large-Linux	2	2	0	m4.large-Linux	1	4	
229192219122	us-west-2	PR003755-pro-bion-asg-0bc0944fbc16a90035c0	normal	Savings Opportunity	Downscale	c4.large-Linux	2	2	0	c4.large-Linux	1	2	
922390019409	us-east-1	mobile-svc-asg-analysisGrid	normal	Savings Opportunity	Downscale	c4.large-Linux	4	16	0	c5.large-Linux	2	8	

Selected:1

Predicted Scaling Activity Resource Utilization Metrics: ASG Effort Details

CPU Utilization - Predicted Day (Current)
PR001514-pro-imco-asg-0bc073c3bc7f780: Workload Chart (Avg 24.35 %)
CPU Utilization - Predicted ...
Time of Day
Min/Max ■ Sustained Activity — High Limit: 85
CPU Utilization - Predicted Day (Recommended)
PR001514-pro-imco-asg-0bc073c3bc7f780: Workload Chart (Avg 24.49 %)
CPU Utilization - Predicted ...
Time of Day
Min/Max ■ Sustained Activity — High Limit: 80
CPU Utilization - Busiest Day (Current)
PR001514-pro-imco-asg-0bc073c3bc7f780: Workload Chart (Avg 27.63 %)
CPU Utilization - Busiest Da...
Time of Day
Min/Max ■ Sustained Activity — High Limit: 85
Memory Utilization in Percent (Current)
PR001514-pro-imco-asg-0bc073c3bc7f780: Workload Chart (Avg 72.73 %)
Memory Utilization in Percen...
Time of Day
Min/Max ■ Sustained Activity — High Limit: 95
Network I/O Bytes (Bytes/s)

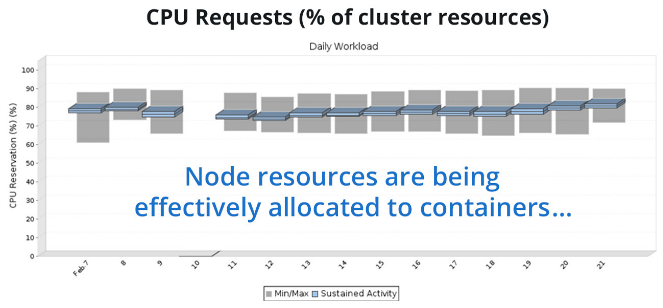
CapOps for Containers

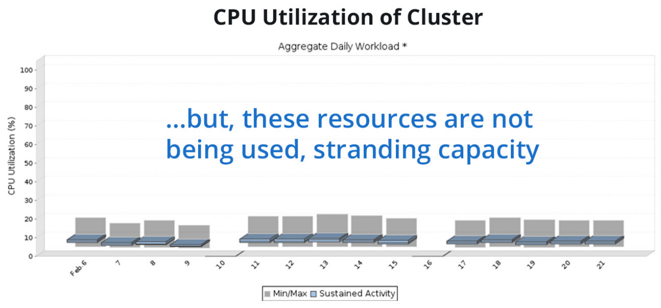
If dealing with the granularity of purchasing in cloud environments creates a resource challenge, then the operational model of containers takes this to an entirely new level. Containers can be even more dynamic, and far more granular, often creating an order of magnitude more entities that must be optimized. In many ways it is like transitioning from the VM-level management to process-level management, and each individual workload, such as a web server or queue manager, must be assigned specific resources. To make things even more complicated, these containers can be combined into pods, replica sets, deployments and other structures, which can be launched from a single manifest, and all of these structures can be governed by various quotas to control resource usage.

And as with the cloud, these characteristics are both a blessing and a curse. Containers have undeniable benefits when it comes to the flexibility and agility they provide when deploying new applications and services. But many people mistakenly believe that they will magically optimize themselves when it comes to resources. This is not the case, and providing inaccurate resource specifications can actually lead to tremendous inefficiency, with resources being stranded and node utilization very low.

Part of this misconception is the fact that containers don't overcommit resources in the same way virtual environments do, meaning that cluster administrators cannot simply tune overcommit ratios to get higher density. The resources assigned to containers are not virtual resources at all, they are ***actual*** resources, meaning they cannot be given out to multiple consumers at the same time.

This removes a key weapon in the battle against inefficiency, and any over-specification of resources translates directly into the need for more infrastructure, either on prem or in the cloud, directly impacting cost.





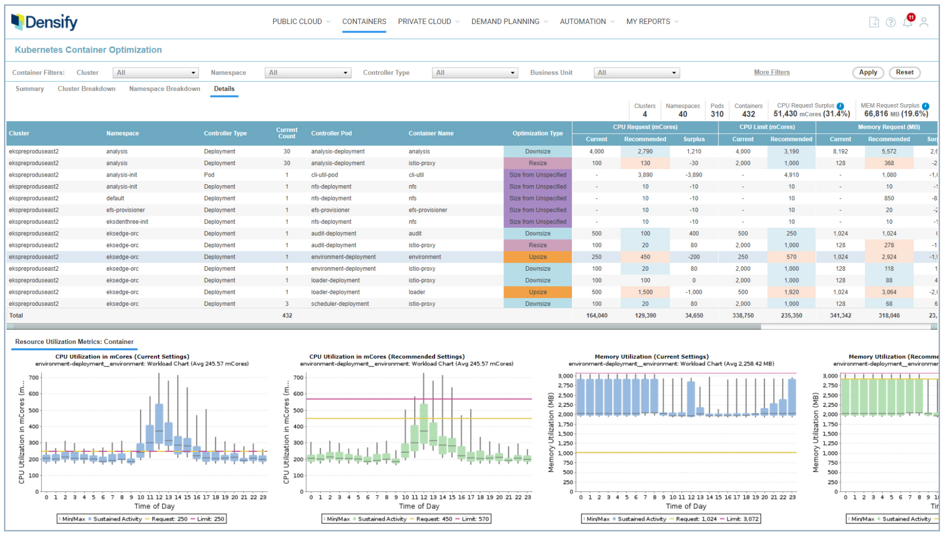
There are also a number of other misconceptions when it comes to containers. If containers are very small then many assume that poor resource specifications couldn't possibly cause high costs, since each container is so insignificant. But if thousands of poorly configured containers are running then this can add up to a tremendous amount, and this is often observed to be the case when analyzing container environments. Similarly, if containers or microservices run for a very short length of time then it is also often assumed that this is relatively harmless. But again, if the services run thousands of times than the error adds up. This "fallacy of insignificance," combined with the impractical amount of effort it would take to manually optimize each container, causes many container environments to be very inefficient.

To combat this, the more operationalized form of capacity optimization provided by CapOps also help address the gap in container resource optimization. This includes:

- **CPU request optimization:** This is amount of CPU resources (in "millicores") guaranteed to a container. The container scheduler must ensure that there are sufficient resources to meet the request values of all containers on a node, so if this value is too high (which is common) then the scheduler will need to spread the containers across more nodes than is necessary, and utilization will be low.
- **CPU limit optimization:** This is the maximum amount a container can consume and setting it too low will cause the scheduler to throttle the performance of a container
- **Memory request optimization:** This is the amount of memory (in megabytes) allocated to a container. Like CPU, if this value is too high then resources will be stranded, and workload density will be low. But, unlike CPU, if this value is too low, then the scheduler may end up placing too many containers on a node, and when the aggregate memory utilization of that node goes above the requested values, the scheduler will actually kill containers to free up resources. This "Out of Memory Killer" (or "OOM killer") is very dangerous, and can be avoided with proper resource optimization.
- **Memory limit optimization:** This is the maximum amount of memory a container can consume, and if it is too low, then this can also cause containers to be killed when their utilization exceeds their limit.

By performing this analysis at the container-level, the results can be then associated back to the Pods, ReplicaSets, and Deployments that they are part of, enabling the optimization to be embedded in the manifests that created the containers. This provides seamless automation, and prevents humans from having to manually deal with optimizing thousands of containers, which simply isn't viable as container environments grow.

Of course, these containers run on nodes, and the optimization of those nodes is also critical. For containers running in cloud environments the nodes are usually cloud instances, typically running in scaling groups, and optimization equates to the cloud instance optimization described above. And because the container and node configurations affect each other, the two forms of optimization must be done together to ensure that the nodes are constantly aligned with the needs of the containers. For example, if container CPU request values are reduced, then memory will typically become the primary constraint in a cluster, and it might be necessary to transition to memory optimized nodes to maintain efficiency. For on-premises environments, the nodes can be VMs (optimized via sizing and placement) or bare metal, requiring long-term purchase planning consistent with an on prem capacity management practice.


Densify
PUBLIC CLOUD
CONTAINERS
PRIVATE CLOUD
DEMAND PLANNING
AUTOMATION
MY REPORTS
Summary
Cluster Breakdown
Namespace Breakdown
Details
More Filters
Apply
Reset
CPU Utilization in mCores (Current Settings)
environment-deployment__environment: Workload Chart (Avg 245.57 mCores)
CPU Utilization in mCores (Recommended Settings)
environment-deployment__environment: Workload Chart (Avg 245.57 mCores)
Memory Utilization (Current Settings)
environment-deployment__environment: Workload Chart (Avg 2,258.42 MB)
Memory Utilization (Recomme
Time of Day
Min/Max
Sustained Activity
Request: 250
Limit: 250
Request: 450
Limit: 570
Request: 1,024
Limit: 3,072

Kubernetes Container Optimization

Container Filters: Cluster: All; Namespace: All; Controller Type: All; Business Unit: All

Clusters 4 | Namespaces 40 | Pods 310 | Containers 432 | CPU Request Surplus 51,430 mCores (31.4%) | MEM Request Surplus 66,816 MB (19.6%)

Cluster	Namespace	Controller Type	Current Count	Controller Pod	Container Name	Optimization Type	CPU Request (mCores)			CPU Limit (mCores)		Memory Request (MB)		
							Current	Recommended	Surplus	Current	Recommended	Current	Recommended	Sur
ekspreproduseast2	analysis	Deployment	30	analysis-deployment	analysis	Downsize	4,000	2,790	1,210	4,000	3,190	8,192	5,572	2,6
ekspreproduseast2	analysis	Deployment	30	analysis-deployment	istio-proxy	Resize	100	130	-30	2,000	1,000	128	368	-2
ekspreproduseast2	analysis-init	Pod	1	cli-util-pod	cli-util	Size from Unspecified	-	3,890	-3,890	-	4,910	-	1,080	-1,
ekspreproduseast2	analysis-init	Deployment	1	nfs-deployment	nfs	Size from Unspecified	-	10	-10	-	10	-	10	-1
ekspreproduseast2	default	Deployment	1	nfs-deployment	nfs	Size from Unspecified	-	10	-10	-	10	-	850	-8
ekspreproduseast2	efs-provisioner	Deployment	1	efs-provisioner	efs-provisioner	Size from Unspecified	-	10	-10	-	10	-	20	-2
ekspreproduseast2	eksdenthree-init	Deployment	1	nfs-deployment	nfs	Size from Unspecified	-	10	-10	-	10	-	10	-1
ekspreproduseast2	eksedge-orc	Deployment	1	audit-deployment	audit	Downsize	500	100	400	500	250	1,024	1,024	[illegible]
ekspreproduseast2	eksedge-orc	Deployment	1	audit-deployment	istio-proxy	Resize	100	20	80	2,000	1,000	128	278	-1
ekspreproduseast2	eksedge-orc	Deployment	1	environment-deployment	environment	Upsize	250	450	-200	250	570	1,024	2,924	-1,
ekspreproduseast2	eksedge-orc	Deployment	1	environment-deployment	istio-proxy	Downsize	100	20	80	2,000	1,000	128	118	1
ekspreproduseast2	eksedge-orc	Deployment	1	loader-deployment	istio-proxy	Downsize	100	100	0	2,000	1,000	128	88	4
ekspreproduseast2	eksedge-orc	Deployment	1	loader-deployment	loader	Upsize	500	1,500	-1,000	500	1,920	1,024	3,064	-2,
ekspreproduseast2	eksedge-orc	Deployment	3	scheduler-deployment	istio-proxy	Downsize	100	20	80	2,000	1,000	128	68	6
Total			432				164,040	129,390	34,650	338,750	235,350	341,342	318,046	23,

Resource Utilization Metrics: Container

Taking Action

Although simply knowing that optimization is required, and quantifying the costs or risks that exist, can be a useful end in itself, the true goal is typically to take action to actually improve the running environment. But this can be challenging, and even organizations that recognize the need for this type of optimization can fail to make a difference if they don't take the right approach to actioning the recommendations. Application owners and lines of business are understandably concerned with the stability of their applications, and will not allow changes to their environments without an air-tight justification and a significant amount of supporting detail.

In order to ensure that the actions generated by a CapOps system met this high bar, and can actually be taken, there are a number of key requirements that must be met:

1. **Precision:** Any recommendation that is generated must be accurate, and account for the minute details that impact the applications. For example, if an app requires local storage, then any recommendation to move to an instance type that doesn't have local storage is useless. If an app is 32-bit, then any recommendation to move from an M3 to an M5 is a non-starter. And, more commonly, if app components have specific resource requirements dictated by the vendor, such as SAP module that must be configured with a specific amount of memory, then any recommendations to downsize these instances are counterproductive. A CapOps system must have detailed policies

to account for this level of detail, and must also use benchmarks to model the impact of changing instance types, in order to provide sufficient precision to enable action. Without this, an organization will not succeed in promoting change, and trying to action flawed recommendation will have the perverse effect of creating more work for subject matter experts as they need to review and vet each recommendation. The last thing you want is a database full of recommendations you can't take.

2. **Integration:** The recommendations that are generated by a CapOps system need to go somewhere, and in environments with highly distributed stakeholders, they would ideally go to systems that those users already use, rather than making these users log into something new. This includes reporting and business intelligence (BI) systems, change management systems, and even DevOps tooling and pipelines, where automation can occur. To support this, recommendations need to exist in both machine-readable and human-readable form, enabling socialization as well as automation. For example, it is very important to have impact analysis reports that provide details of a recommended change (including the predicted utilization impact). These can be attached to change tickets, or distributed through messaging systems, and have a tremendous impact on the willingness to approve those changes. Similarly, business group rankings and "shameback" reports are also useful in promoting action, by providing transparency across the business.

3. **Automation:** Although success can be had without going to full automation, it is typically the long-term goal for many organizations, particularly as they achieve scale and move to containers. As the number of "moving parts" that must be optimized increases, manual action becomes less and less viable, and the risk of human error becomes higher and higher. But any automation strategy must also adhere to change management requirements, and the ideal solution is one that provides transparency (e.g. app owner reports), change control (e.g. ITSM integration), and full automation of the change when approval is attained. With sufficient trust in the analytics (consistent with the "precision" requirement) some organizations have negotiated with app teams to remove the approval requirement, which greatly streamlines the automation process.

In addition to these three requirements, a CapOps system must also be open, allowing access to the data and recommendations in order to feed other tools in the ecosystem. Because collecting data in cloud and container environments can be a challenge, a CapOps system that contains all of this data can be valuable on this basis alone. But combining this raw data with optimization analysis results and associated metadata is even more powerful, and a "Resource Management DB" (or "CapOpsDB") that contains all of this data could well become a key component in future tooling architectures.

The rise of tools like Grafana is evidence of the need for this kind of component, and expanding the data available to these tools to include detailed optimization results and predictive analysis models would be a logical progression for most organizations. It is also consistent with the move toward observability, and by combining the CapOps data with logs, tracing, performance analysis and other "data lakes" the combination can become greater than the sum of its parts. Regardless of whether an organization takes a "centrally-managed" versus a "centrally-coordinated" approach to capacity, it always makes sense to start with a "centrally-analyzed" set of answers.

Conclusion

Using history as our guide, and following the evolution of DevOps and FinOps, it is logical that CapOps, or something like it, will emerge to pick up the capacity torch that was temporarily dropped in the move to cloud. Reading the bill, and purchasing commitment-based discounts, will only take an organization so far, and the next step is to optimize the actual resources that are purchased. And focusing on the elasticity inherent in the cloud and container environments, and making sure the cloud-native constructs are working like a well-oiled machine, will make an organization much more responsive to changing business needs, less apt to experience operational issues, and far less likely to experience high cloud bills that are not reflective of their true needs.