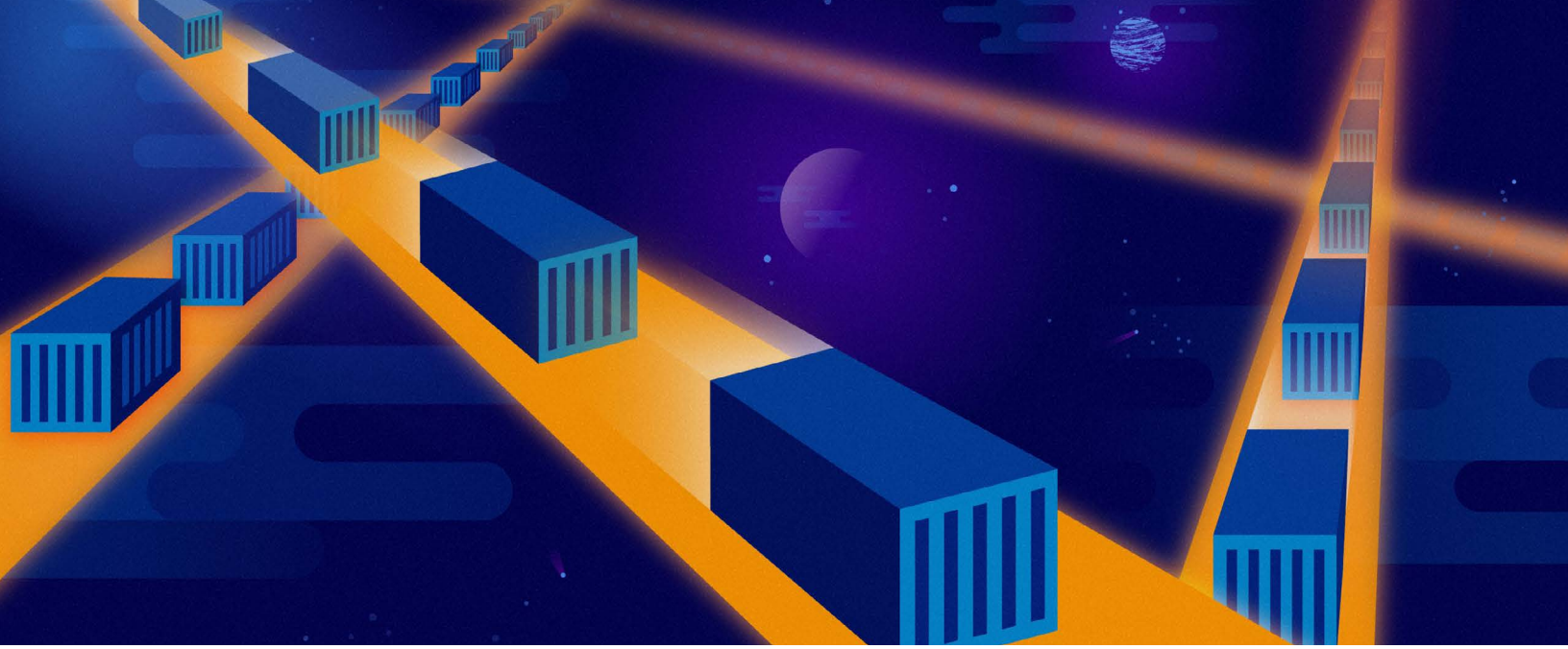
Cloud-scale monitoring with AWS and Datadog

Containerized applications in AWS

aws | DATADOG

Cloud-scale monitoring with AWS and Datadog

Containerized applications in AWS



The evolution of containers in the cloud

Containers power many of the applications we use every day. Particularly well suited for microservice-oriented architectures and agile workflows, containers help organizations improve developer efficiency, feature velocity, and optimization of resources.

In this section, we'll take a look at the changing state of containers in the cloud and explore why orchestration technologies have become an essential part of today's container ecosystem. Then, we will discuss the key metrics to monitor when leveraging two container orchestration systems: Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS). Finally, we will explore how you can collect all of those metrics and get full-stack visibility into containerized environments with a comprehensive monitoring platform like Datadog.

Ever-changing containerized workloads

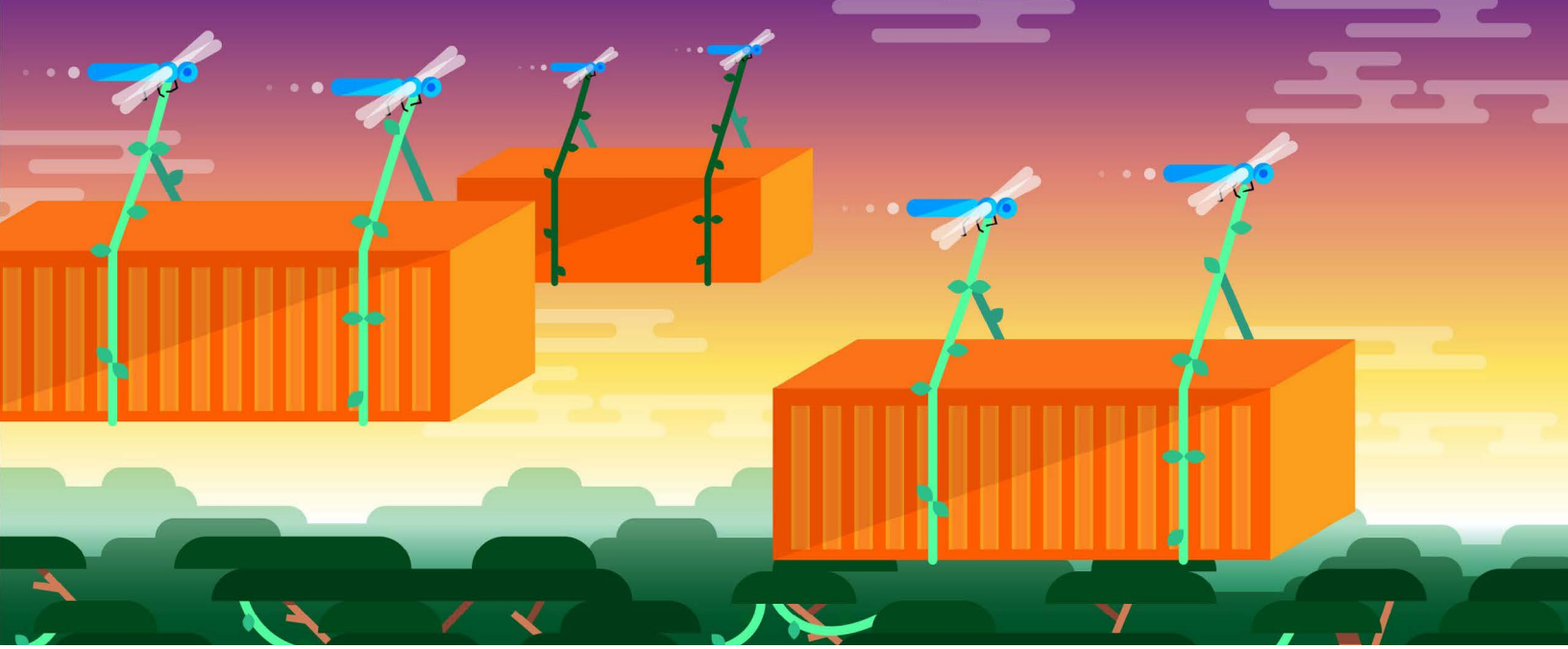
At its most basic level, a container is an isolated virtual runtime that comes packaged with all of the dependencies it needs. This simplifies the development process by helping ensure that applications run consistently everywhere, whether they're deployed in a local environment, your company's private data center, or in the public cloud. Containerization involves breaking down your applications into isolated units of software that are more lightweight than virtual machines (VMs) and easier to deploy and scale, which leads to more efficient provisioning of resources and helps support agile workflows and rapid release cycles.

As container environments grow in complexity, development teams have increasingly turned to container orchestration solutions such as Kubernetes to help facilitate the work of managing, scaling, and operating dynamic container infrastructure (e.g., deploying containers on nodes that have enough available resources, restarting unhealthy containers, ensuring that containers are running across all desired regions for high availability, and other tasks).

Orchestration helps teams manage dynamic container environments at scale and use infrastructure resources more efficiently. According to a report of usage data from more than 10,000 companies that have adopted Docker, unorchestrated environments had a median value of 6.5 containers running on each host, compared to 11.5 containers per host in orchestrated environments. This supports the notion that orchestrators can help organizations run more containers and utilize more of the available resources on each host, effectively lowering their infrastructure costs.

To make container orchestration more accessible to a broader range of organizations, cloud providers offer managed services that feature built-in support for high availability, security, and networking, and address many of the operational challenges associated with container orchestration.

This section will focus on two managed container orchestration technologies that are specifically designed to help you run containerized applications effectively in the AWS cloud: Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS). We'll explore an overview of these two container orchestration technologies and the key metrics you should monitor when using them. We will also discuss how AWS Fargate provides serverless infrastructure for these services, eliminating the need to provision or manage your own hosts.



Amazon Elastic Container Service (ECS)

How Amazon ECS works

First released in 2015, **Amazon Elastic Container Service (ECS)** helps users efficiently manage and scale containers in the AWS cloud. ECS allows you to launch containers on Amazon EC2 infrastructure, AWS Fargate–managed compute resources, or a combination of both. While deploying containers on EC2 infrastructure gives you more direct control over the servers that host your containers, AWS Fargate provides the convenience of not having to provision, manage, or monitor the infrastructure that runs your containers.

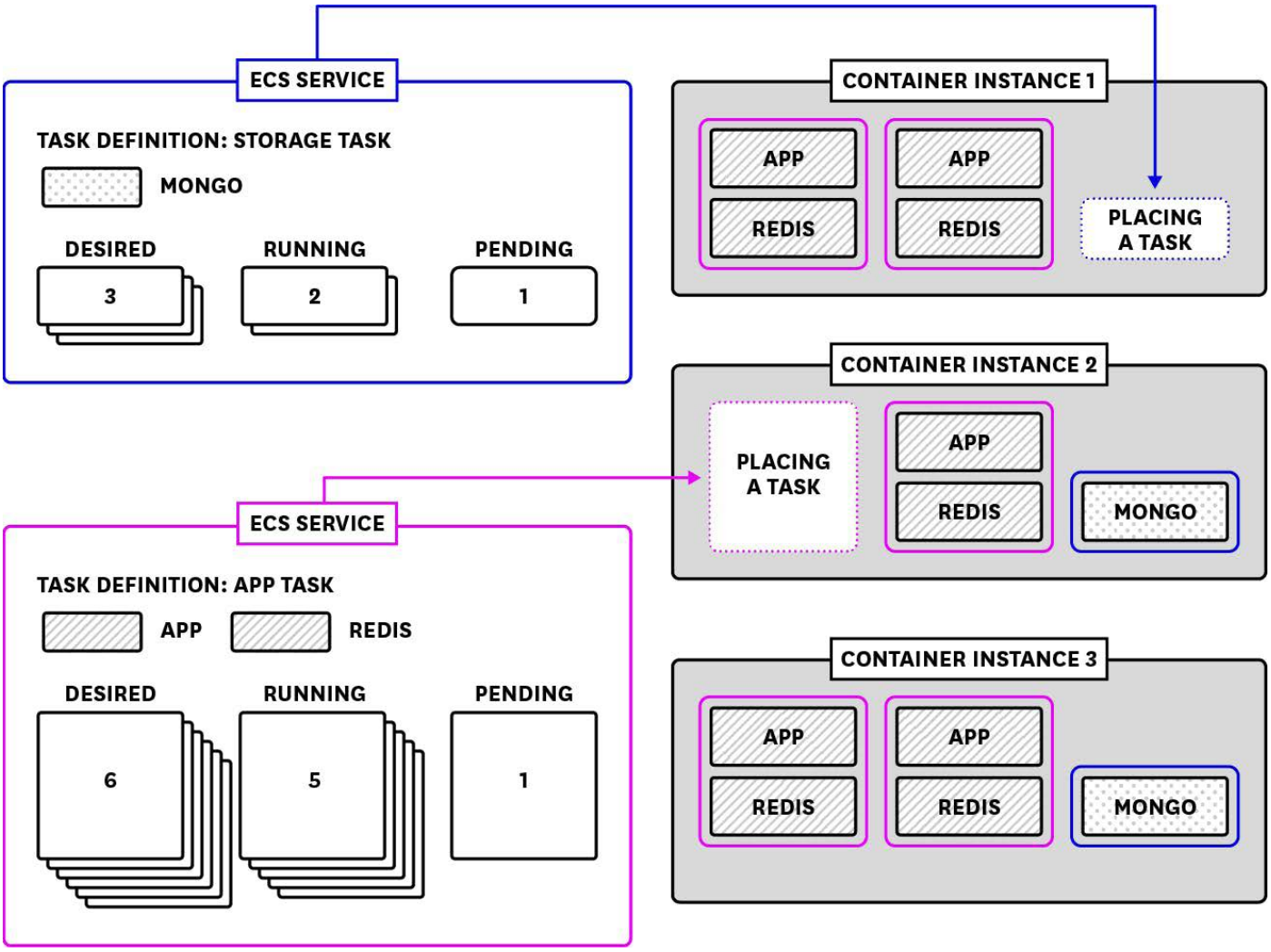
Tasks serve as the foundation of ECS workloads. Each task is responsible for launching, terminating, and configuring containers according to its **task definition**—instructions that may describe which container images to run, how many resources to allocate to each container, and other information. Task definitions can name images from Amazon Elastic Container Registry (ECR), Docker Hub, or any other registry.

You can create a **service** to automatically schedule a task according to the specifications outlined in its task definition. Within the service definition, you can specify the desired number of instances of a task to run. The service will continuously track the status of the task and launch and terminate tasks accordingly, which helps ensure that you are running the desired number of task instances at any given time.

You can determine what type of infrastructure (e.g., EC2, Fargate, or a combination) you want your ECS tasks to use by configuring a capacity provider strategy or specifying the launch type in a task definition. For example, the following task definition specifies a Flask application container and a Redis container. The `requiresCompatibilities` parameter indicates that this task should use the Fargate launch type.

```
{
  "family": "my-flask-app-family",
  "executionRoleArn": "arn:aws:iam::<ACCOUNT_ID>:role/ecsTaskExecutionRole",
  "compatibilities": [
    "EC2",
    "FARGATE"
  ],
  ""containerDefinitions": [
    {
      "entryPoint": [
        "python",
        "app.py"
      ],
      "essential": true,
      "image": "my-flask-app",
      "name": "app",
      "portMappings": [
        {
          "containerPort": 4999,
          "hostPort": 4999,
          "protocol": "tcp"
        }
      ]
    },
    {
      "essential": true,
      "image": "redis:latest",
      "name": "redis",
      "portMappings": [
        {
          "containerPort": 6379,
          "hostPort": 6379,
          "protocol": "tcp"
        }
      ]
    },
  ],
  "cpu": "256",
  "memory": "512",
  "networkMode": "awsvpc",
  "requiresCompatibilities": [
    "FARGATE"
  ],
  "revision": 11,
  "status": "ACTIVE"
}
```

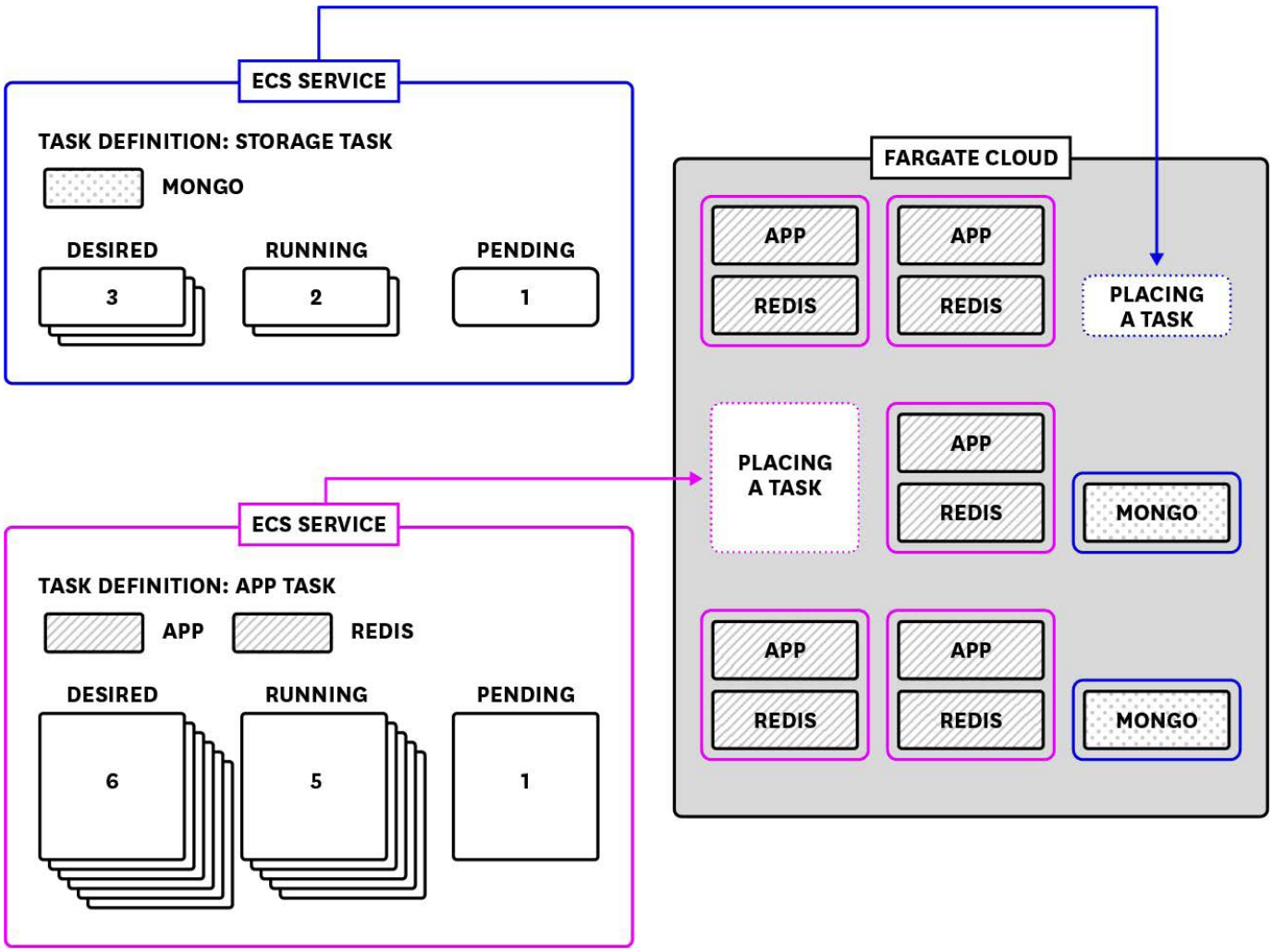
ECS ON EC2 INFRASTRUCTURE



Deploying ECS tasks to Amazon EC2 can give you more flexibility to provision, manage, and scale the virtual instances that run your workloads. In this setup, each EC2 instance—known as an ECS container instance—runs the ECS container agent, which is responsible for managing the containers that are running on that instance. The agent informs the ECS API about running tasks and manages tasks in response to requests from ECS.

For tasks that run on EC2 infrastructure, the amount of resources available is determined by the resource requirements specified in the task definition. If resources are not configured in the task definition, resources are automatically limited by the capacity of the container instance that hosts the task. Monitoring ECS resource metrics is critical for creating suitable scaling policies for your ECS services and ensuring that your workloads have enough capacity to run smoothly. We will cover these and other metrics in more detail in a later section.

ECS ON FARGATE



Fargate eliminates the need to manage or provision EC2 instances by placing ECS tasks on AWS-managed infrastructure. ECS automatically provisions Fargate compute resources, sized according to the resource requirements in the task definition. Whereas ECS tasks could fail to get scheduled if EC2 container instances do not have enough available resource capacity to run them, Fargate takes the uncertainty out of the equation so that engineering teams can spend less time juggling operational concerns.

Regardless of the type of infrastructure you're using in ECS, you'll want to keep an eye on the following key metrics to ensure that your containerized applications are running smoothly.

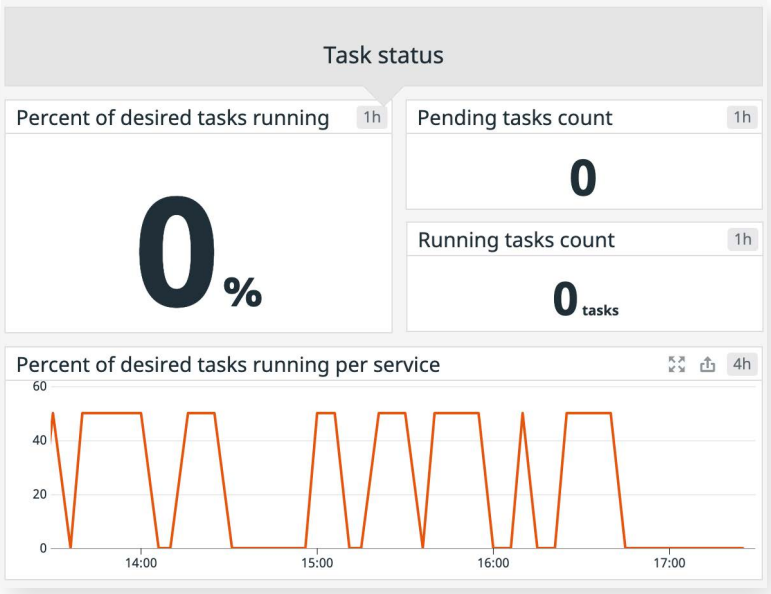
How to monitor Amazon ECS

As you run automated deployments in Amazon ECS, you'll want to monitor the status of your cluster to ensure that containers are being launched and terminated as expected. You'll also want to monitor the resource usage of your ECS workloads—and, if you're not using Fargate, any container instances they're running on.

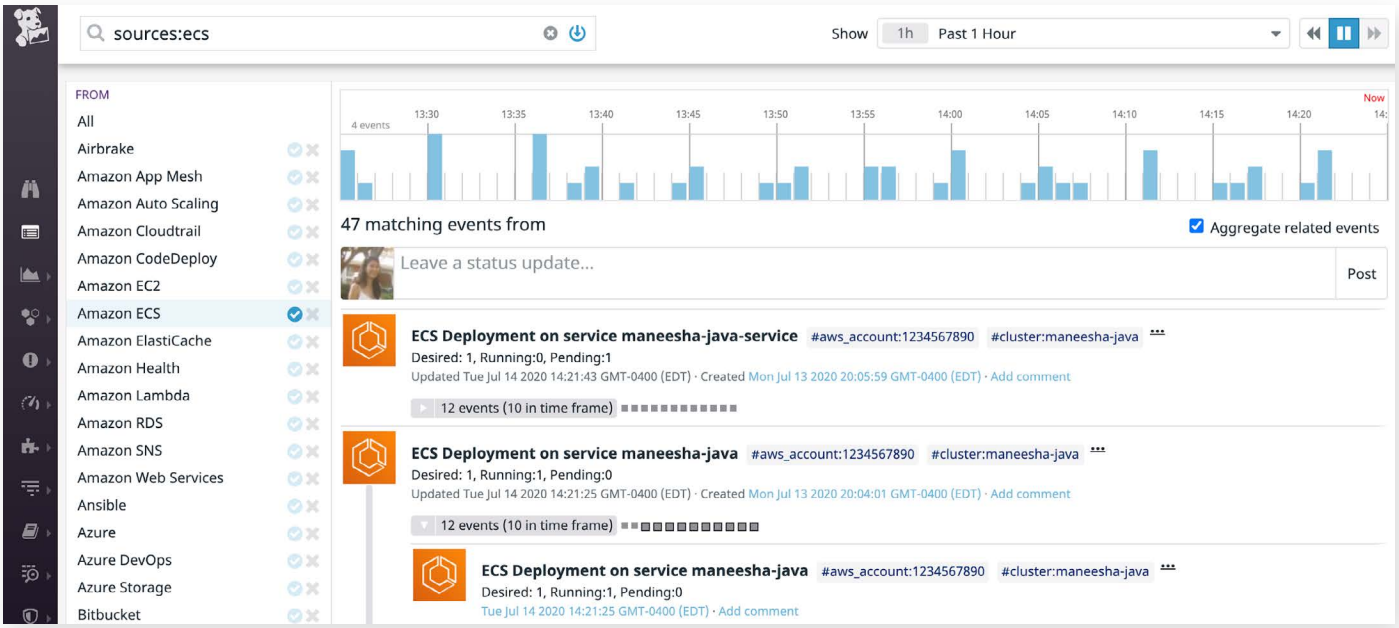
MONITORING ECS STATUS

Desired task count vs. running task count per service: ECS services automatically ensure that the desired number of tasks is running at all times. If a task unexpectedly stops or fails, the running task count will fall below the desired task count until the ECS service launches another task to replace it.

If you've recently deployed a new version of a service that increases the desired instances of a task, the ECS service should automatically launch tasks to meet the new requirements. This can result in a temporary discrepancy between the desired task count and running task count as new tasks are initializing. However, if it looks like the running task count is consistently lower than the desired task count, you should investigate to see why those tasks are not running.



For example, the latest task definition could contain a typo, a task may have failed an Elastic Load Balancing health check, or the container instance running the task might have been stopped or terminated. In addition to comparing the desired and running task count, service events can help you understand if ECS is running into any issues with placing tasks in your cluster.



KEY ECS RESOURCE METRICS TO MONITOR

ECS task definitions can specify the maximum amount of resources that ECS will allocate to a task and/or individual containers. Task-level resource limits serve as the maximum amount of resources that ECS will make available for all containers in the task to share. You may also specify resource limits for individual containers within a task in order to manage resources on a more granular level.

If you are using Fargate, you are required to configure a task-level resource limit, also known as a task size. ECS uses this to determine the amount of compute resources that it will provision for the task, which is also the maximum amount of CPU and memory that a task will be able to use. For tasks that get deployed to EC2 infrastructure, task-level resource limits are optional because the maximum amount of resources available to the task is automatically limited to the resource capacity of the container instance it runs on.

ECS tasks running on EC2 require that you specify a container-level memory limit. Container-level CPU limits, however, are optional because, unless otherwise specified, ECS will reserve a default number of CPU units for each container. You can set hard or soft memory limits (or both) for containers in a task. If you specify a soft memory limit, ECS will use that value to determine how much memory to reserve. Otherwise, if you only specify a hard limit, that value will serve as the amount of memory that ECS reserves. Setting reasonably small soft limits provides ECS with flexibility for scheduling more tasks on fewer container instances, which could provide cost savings. However, it may also lead to performance issues if your containers actually need more resources than they reserved.

Resource reservation metrics are available to help you track the percentage of total resource capacity that is currently reserved by ECS tasks running on EC2. The amount of reserved resources is calculated based on the container-level resource limits defined in your task definitions (or, in the case of CPU, the default number of reserved units if a limit is not specified), whereas the total resource capacity is calculated based on the size of the container instances in the cluster.

Monitoring the following resource usage metrics can help you create effective Auto Scaling policies for workloads on EC2 and Fargate. Metrics that are only applicable to tasks running on EC2 container instances are marked accordingly.

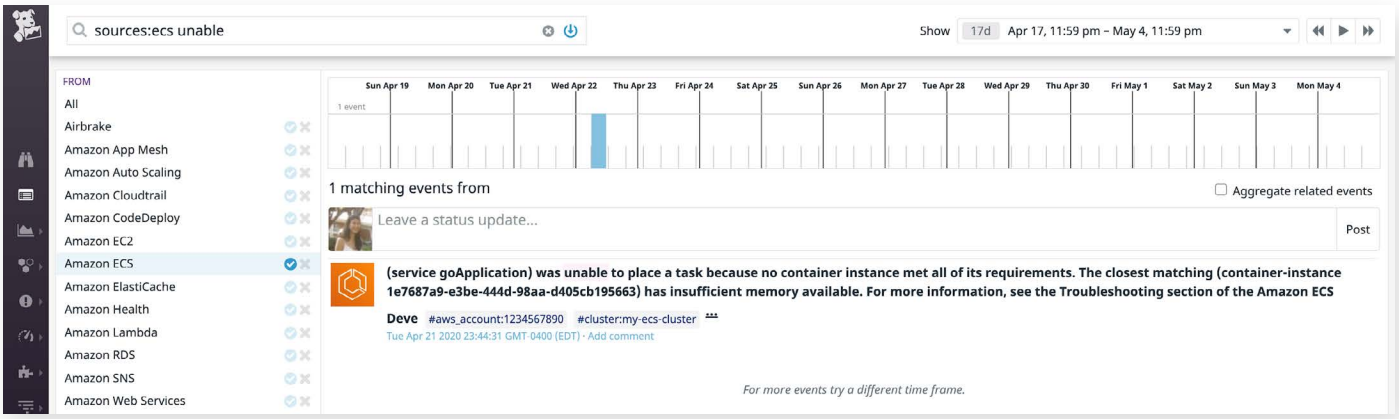
Memory utilization: If you've set a hard memory limit for any container in a task, you may want to alert your team if container-level memory utilization is consistently high, because ECS will terminate any container that crosses its hard limit. You may also want to update your task definitions to either increase or remove this hard limit. If you are using EC2, you can configure swap space for containers within a task definition, which can help reduce the likelihood of OOM-killed processes but may not be a suitable option for latency-critical applications.

CPU utilization: If you are using EC2 to run your tasks, monitoring cluster-level CPU utilization can help you determine if you have enough container instances running, or if you need to add more to support your workloads. In EC2, this metric measures the percentage of reserved CPU that is currently in use across a service or cluster. As such, you may see this metric go above 100 percent if the combined CPU usage of containers in a service exceeds the amount of CPU initially reserved for those containers (note: this is not the case for Windows container instances, where the CPU reservation functions as a hard limit). You can monitor this metric and use it to configure policies that automatically adjust the size of an ECS service based on its real-time CPU utilization.

Regardless of whether you are running ECS tasks on EC2 or Fargate, container-level CPU utilization metrics can be helpful for identifying containers that are particularly CPU-intensive. For example, you may determine that you need to configure container-level CPU limits to prevent certain containers from hogging resources that other containers in the task may need to complete their work.

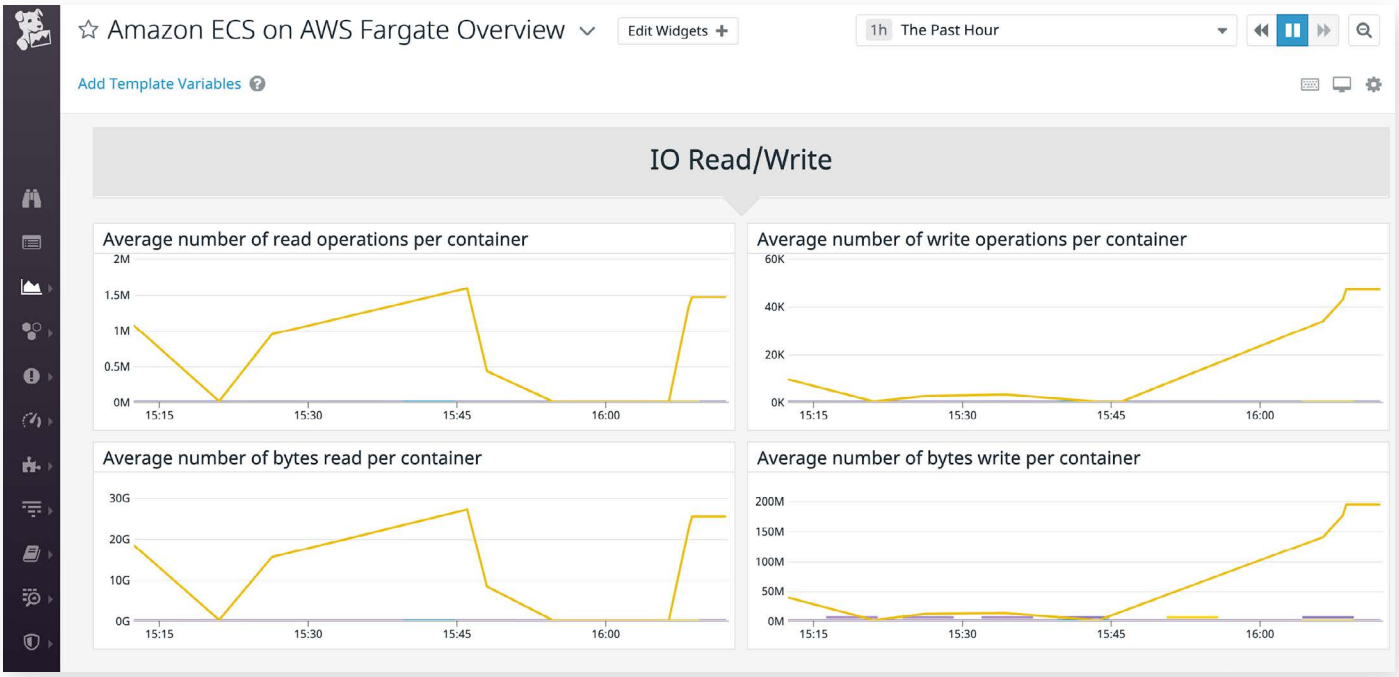
CPU and memory reservation (only on EC2): Reservation metrics only apply to ECS tasks running on EC2 container instances. CPU and memory reservation metrics measure the percentage of the total CPU or memory (calculated based on the combined capacity of all instances in the cluster) that is reserved by running tasks. Monitoring these metrics can help ensure that ECS is able to schedule and launch your workloads successfully.

For instance, if ECS is unable to find a container instance that has enough available memory to run a task, the task will remain in pending state. An ECS service event will report a message similar to the following: "service <SERVICE_NAME> was unable to place a task because no container instance met all of its requirements." To fix this issue, you could reduce the amount of memory reserved in the task definition or add more container instances that have enough memory to accommodate the task's resource requirements.



Resource capacity issues could help explain why the running task count remains below the desired task count. To prevent this from happening, you can create an Auto Scaling policy that provisions more container instances if memory or CPU reservation rises above a certain level.

I/O metrics: In ECS, you can monitor unexpected drops in the volume of bytes read to or written from each EC2 container instance or container to ensure that your tasks are able to access the storage they need. ECS tasks can use Amazon Elastic File System (EFS) volumes to provide containers with access to highly scalable, persistent storage. Alternatively, you can specify other storage options like bind mounts (e.g., mounting an Amazon Elastic Block Store volume that is already attached to your container instance).



Because ECS does not consider the disk requirements of a task before placement, it is possible that tasks will run on container instances that do not meet their storage requirements. One advantage of using EFS is that you do not need to monitor the amount of storage remaining on your volumes—EFS will automatically scale as you add more data to your cluster. If your EC2 tasks are not using EFS, you should monitor disk space available on your container instances and move workloads over to new instances that have enough storage.

Network throughput: Monitoring the status of network connections within your cluster—not only between your own containerized microservices but also between the ECS container agent and the ECS API—is crucial. You can set up an alert to detect issues with network throughput going in and out of your container instances (if applicable) and containers, as this may point to a configuration issue. If you are using Application Load Balancers (ALBs) to distribute traffic across tasks in your ECS services, ALB metrics and logs can also help you investigate issues that are preventing your services from receiving traffic.

Amazon Elastic Kubernetes Service (EKS)

Kubernetes, or K8s, is a popular container orchestration solution that was open sourced by Google in 2014. Although many organizations run self-managed Kubernetes clusters in AWS, Kubernetes infrastructure can be challenging to operate without technical teams dedicated solely to this purpose. Amazon Elastic Kubernetes Service (EKS) enables organizations to access all of the features that are built into Kubernetes while reducing their operational overhead.

How Amazon EKS works

Amazon EKS provides a fully managed Kubernetes control plane, which is responsible for maintaining the state of the cluster, managing and scheduling workloads, and other critical tasks. The control plane is composed of the following components:

- The **API server**, which exposes the API through which clients/applications and worker nodes can access information about the cluster
- The **controller manager**, which is responsible for moving the cluster closer to its desired state
- **Schedulers** that query the state of the cluster and determine how workloads should get assigned to worker nodes
- **etcd**, a distributed key-value store that tracks the state of the cluster, cluster configuration data, and other information

The control plane functions as the brain of the cluster, so keeping it healthy and highly available is crucial. Although you can run the control plane on a single node, provisioning at least three control plane nodes is recommended. Amazon EKS deploys control plane nodes across multiple AWS availability zones to ensure high availability. It also continuously monitors the control plane nodes and replaces them if they are unhealthy.

In addition to a fully managed control plane, EKS provides built-in networking for your cluster—not only between the control plane and worker nodes but also between the pods located on each node. And EKS is certified Kubernetes conformant, so you can easily get started, whether you're migrating Kubernetes workloads to EKS or launching a new cluster.

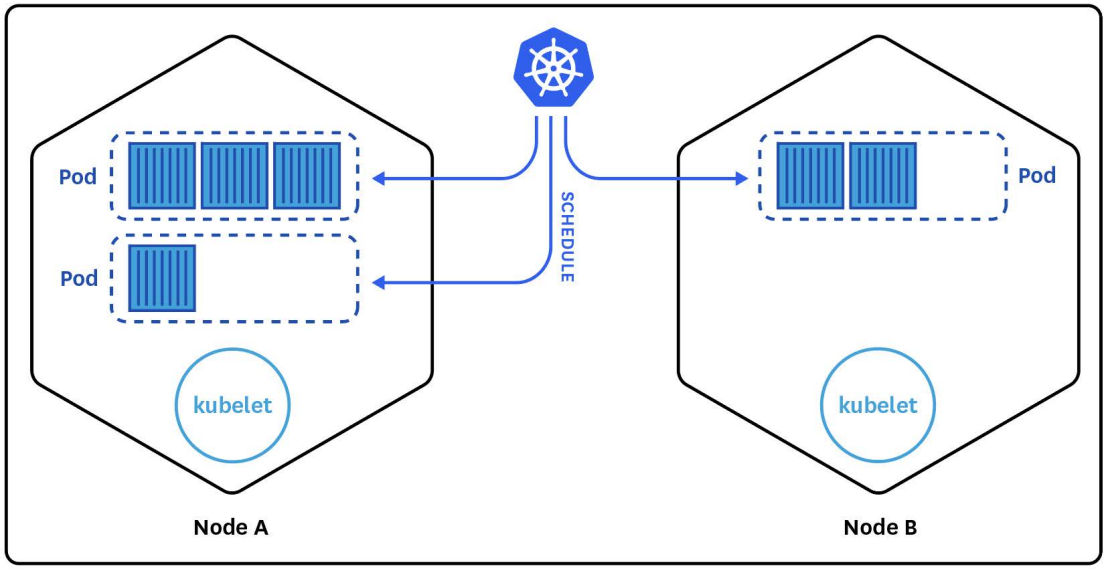
EKS integrates seamlessly with complementary services like AWS App Mesh (for application networking) and AWS CloudFormation (for deploying EKS infrastructure as code) to make it easier to run Kubernetes clusters in the AWS cloud.

HOW EKS MANAGES WORKLOADS

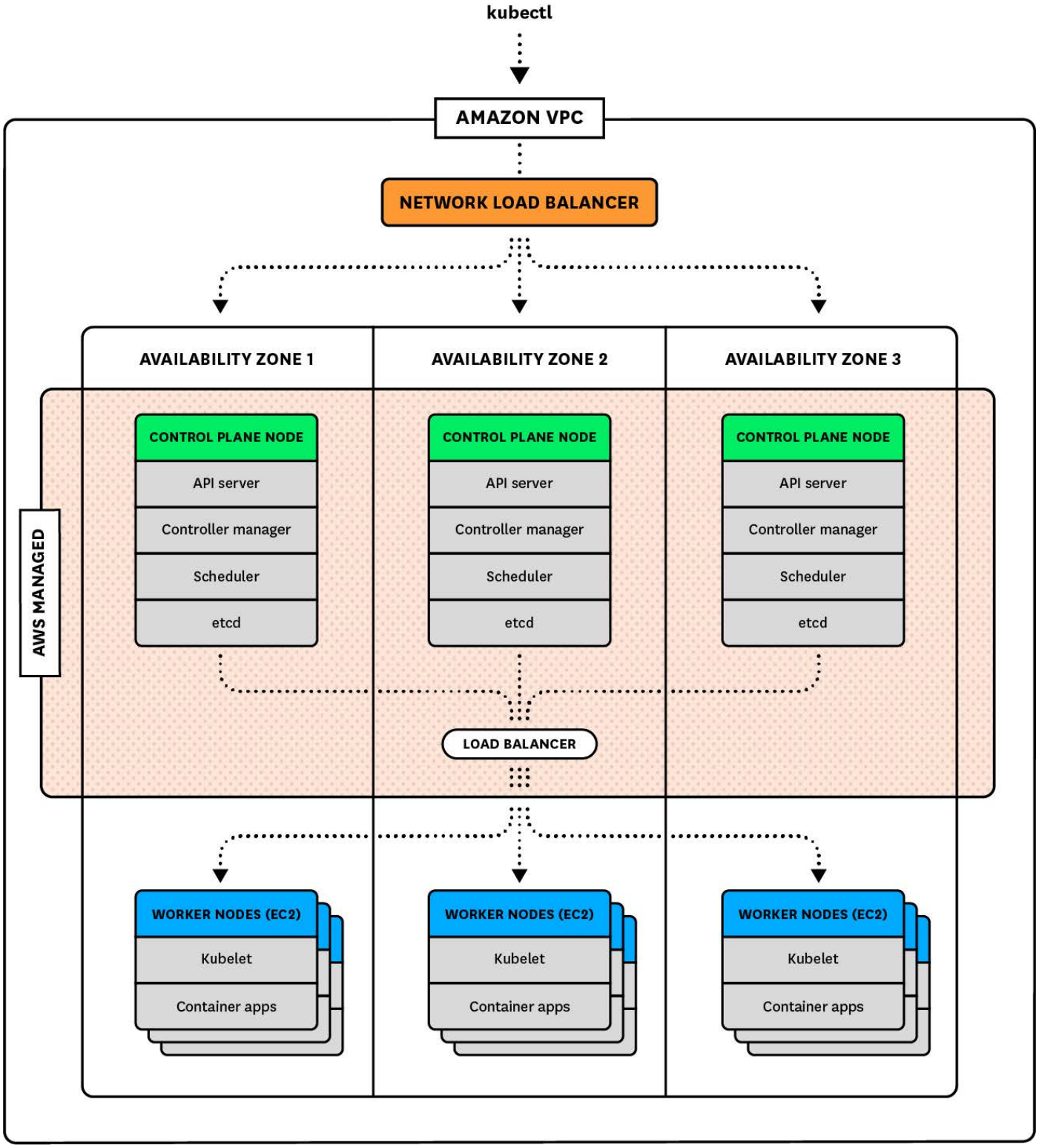
An EKS cluster is composed of two types of nodes: control plane nodes (managed by AWS) and worker nodes (either EC2 instances or Fargate-managed compute resources). Each worker node runs a kubelet process that monitors the node and communicates with the control plane. Worker nodes also run one or more **pods**, the smallest units that can be deployed in Kubernetes. Pods are easily scalable groups of one or more containers that run your application workloads. Containers in the same pod are granted access to shared storage and a single network IP. A pod can be any unit that runs a workload in your cluster, whether it's an instance of an application or a single component (such as NGINX).

Manifests are configuration files (in YAML or JSON format) that define the desired state of objects in the cluster. Manifests describe the number of pods to launch and the containers that make up each pod and may also include resource requirements for your workloads.

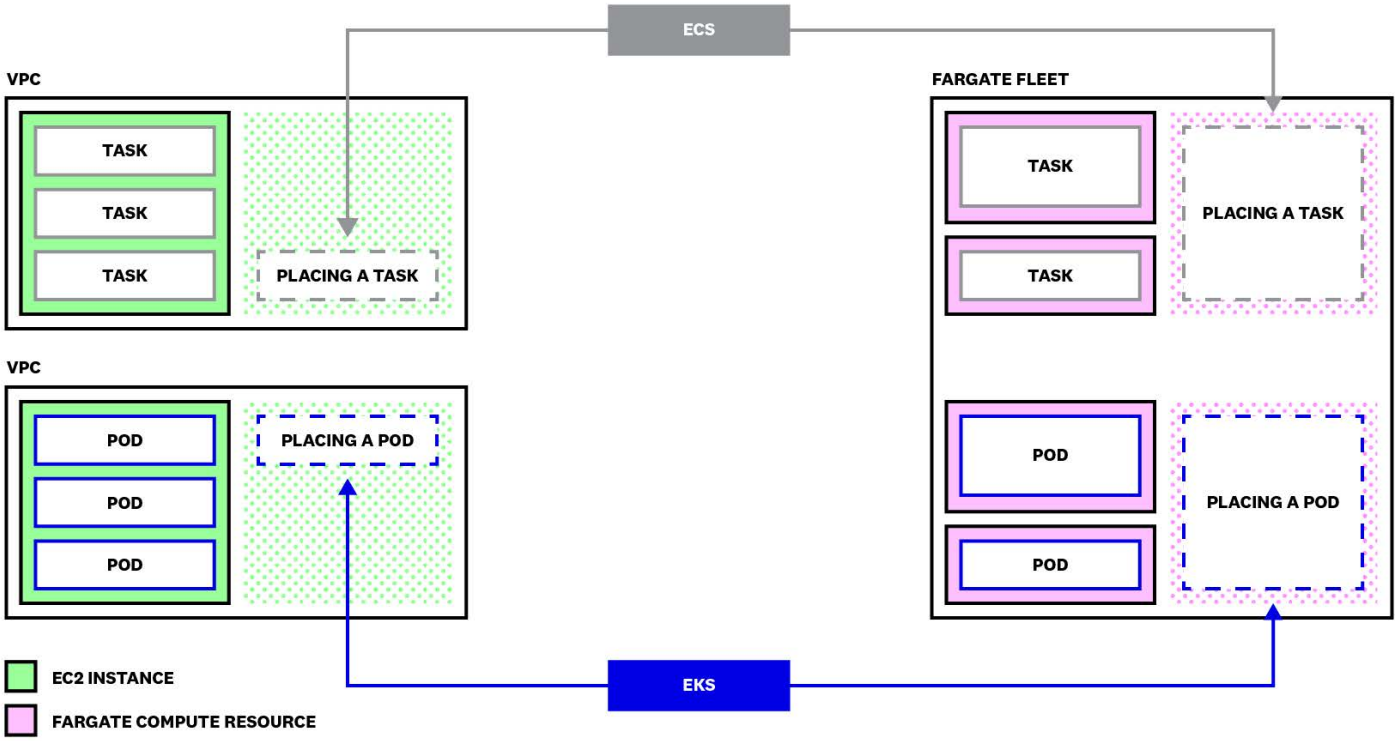
The control plane uses the information in your manifests to determine where it should schedule pods throughout the cluster.



Like ECS, EKS can launch workloads on EC2 worker nodes or on Fargate. If you deploy EKS on EC2 infrastructure, your Kubernetes worker nodes are essentially EC2 instances, as shown in the figure below. EKS runs in a virtual private cloud (VPC) that provides isolation for your cluster.



Whereas EKS removes the need to manage your Kubernetes control plane nodes, EKS on Fargate goes a step further so that you no longer need to manage any worker nodes at all. If an EKS pod meets the specifications outlined in any Fargate profile, AWS will launch the pod on a dedicated, Fargate-managed compute resource that is sized according to the pod's resource specifications. If a pod specification does not include any resource requests, Fargate will default to using the smallest resource specification to provision a compute resource from its fleet.



How to monitor Amazon EKS

As you scale and operate your Amazon EKS workloads, you'll want to monitor the status of the cluster and the control plane to ensure that everything is running smoothly. It's also important to monitor the resource usage across your EKS containers, pods, and (if applicable) hosts so you can ensure that EKS is not evicting pods or throttling your workloads due to resource constraints.

MONITORING EKS STATUS

The Kubernetes API server emits cluster state information about the count, health, and availability of Kubernetes objects, such as pods. Kubernetes uses this data to determine if it needs to launch, schedule, or terminate pods to bring the cluster closer to the desired state.

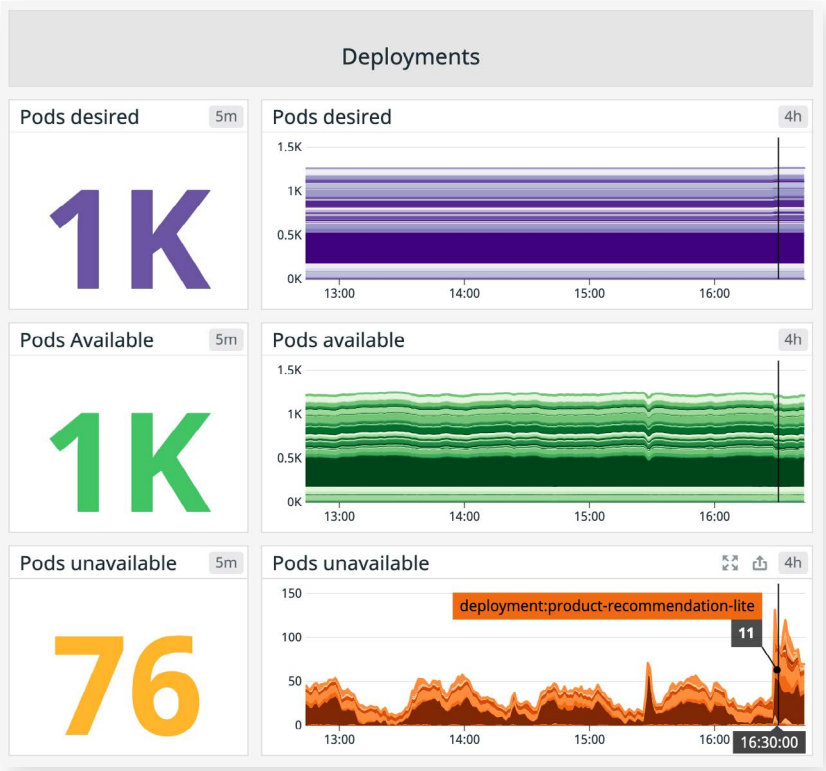
Kubernetes uses **controllers** to manage the state of your cluster. Two main types of controllers are DaemonSets and Deployments. **DaemonSets** (not supported in Fargate) ensure that a particular pod is running on every node in the cluster (or on a specified set of nodes). **Deployments** launch a specific number of pods to run a workload. Because of the ephemeral state of pods, a Deployment manifest typically also defines a Service, which provides persistent access to the pods running in the Deployment. The manifest below defines a Deployment that creates three pods, each running a Redis container. It also defines a Service that allows access to the pods in this Deployment.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
spec:
  replicas: 3
  template:
    metadata:
      labels:
        role: redis
    spec:
      containers:
      - name: redis
        image: redis:5.0
        ports:
        - name: redis
          containerPort: 6379
---
apiVersion: v1
kind: Service
metadata:
  name: redis
  labels:
    role: redis
spec:
  ports:
  - port: 6379
    targetPort: 6379
  selector:
    role: redis
```

Kubernetes emits cluster state metrics that vary based on the controllers you are using (e.g., the desired pod count is measured as `kube_deployment_spec_replicas` in a Deployment vs. `kube_daemonset_status_desired_number_scheduled` in a DaemonSet). Regardless of these small differences, all cluster state metrics give you visibility into the state of objects across the cluster, whether they are running as a DaemonSet, Deployment, or any other type of controller.

The following cluster state metrics can be helpful for surfacing issues with launching workloads and ensuring that your cluster is sized appropriately. Metrics that are only applicable to pods running on EC2 worker nodes are marked accordingly.

Desired vs. current pods: If everything is going smoothly, these numbers should match. Alerting on a prolonged discrepancy in these metrics can help you detect problems, such as configuration errors that are causing pods to fail. Inspecting pod logs can help you get further insight for troubleshooting.

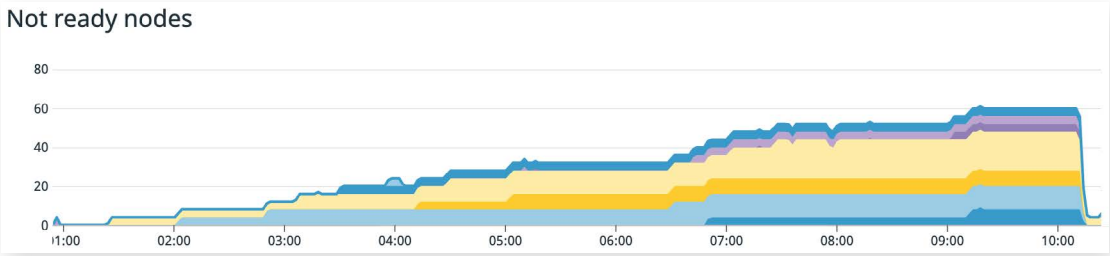


If your cluster is not running the desired number of pods, it could indicate that your nodes are running into resource capacity issues. When EKS deploys pods to EC2 infrastructure, it will only schedule pods on worker nodes that have the available resources to fulfill their resource requests. Pods may end up getting stalled in a pending state when none of your nodes have enough available resources to run them. If so, you may need to revise your cluster autoscaling policies to avoid running into these issues. This is less of a concern when deploying EKS pods to Fargate because AWS provisions dedicated compute resources for pods to ensure that pods' resource requests can be fulfilled.

Node status (only on EC2): This metric only applies if you are using EC2 instances to run your EKS pods. Each EKS worker node emits the following health conditions to the control plane whenever it detects a change in status, or every five minutes by default (though this period is configurable):

- `Ready`: `True` if node is ready to accept pods
- `MemoryPressure`: `True` if node memory is too low
- `PIDPressure`: `True` if too many processes are running
- `DiskPressure`: `True` if remaining disk capacity is too low
- `NetworkUnavailable`: `True` if networking is not properly configured

Monitoring the `Ready` and `NetworkUnavailable` checks can help you detect and troubleshoot issues with nodes that are unable to run pods.

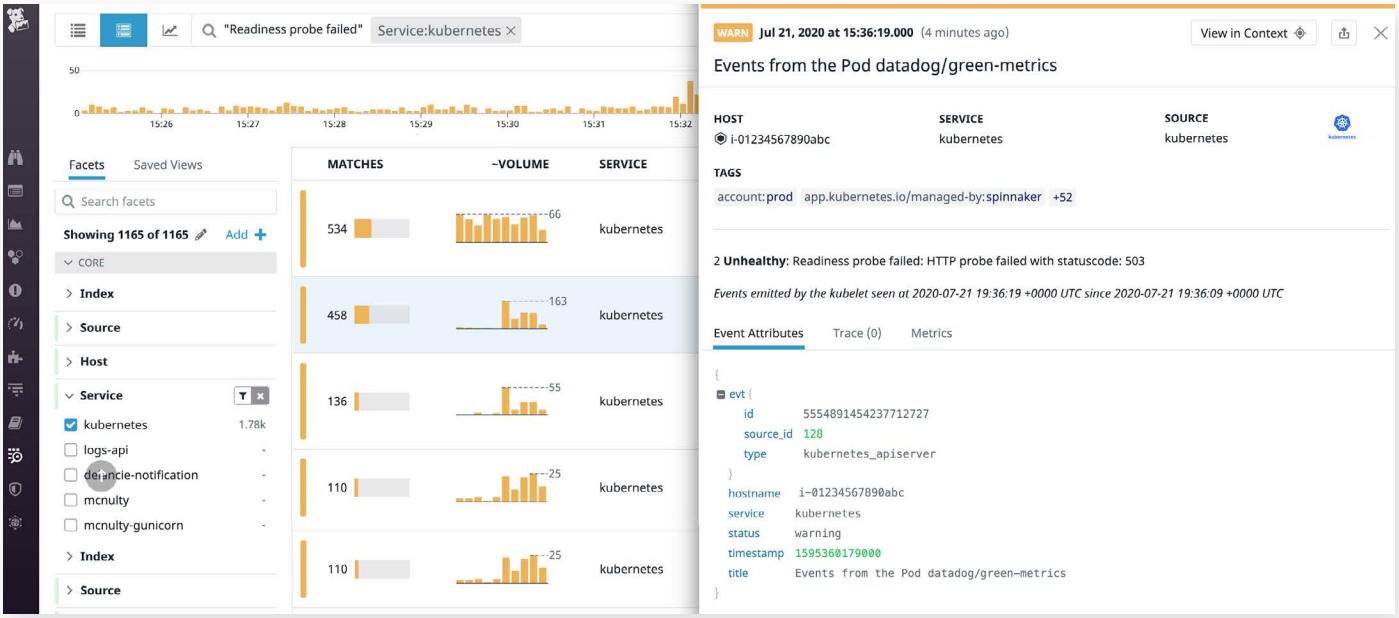


If a node's `Ready` condition returns `Unknown` or `False` for longer than the configured timeout (5 minutes, by default), Kubernetes will delete all pods on that node. If a node returns `True` for the `MemoryPressure` or `DiskPressure` check, the kubelet will try to reclaim resources.

Pod capacity (only on EC2): In EKS, the instance type determines how many elastic network interfaces (ENIs) are supported, and each ENI only supports a finite number of IP addresses. Since each pod has its own unique IP address, any given node or cluster can only support a limited number of pods.

Keeping track of the maximum number of IP addresses—and therefore the maximum number of pods—allowed on each instance type and comparing it against the number of running or desired pods, as well as your kubelets' pod limits (110 pods per node, by default), will help you determine if you need to scale up the number of nodes in your fleet.

Available and unavailable pods: If you detect an increase in unavailable pods—or if certain pods consistently remain unavailable—it could signal a configuration issue. For example, you may have configured a readiness probe to give your containers time to complete certain tasks—such as loading a cache into memory—before they start accepting requests. If a readiness probe is too demanding (e.g., it requires a third-party dependency that isn't strictly necessary), it could prevent your pod from becoming available. Consult your pods' logs to get more details.

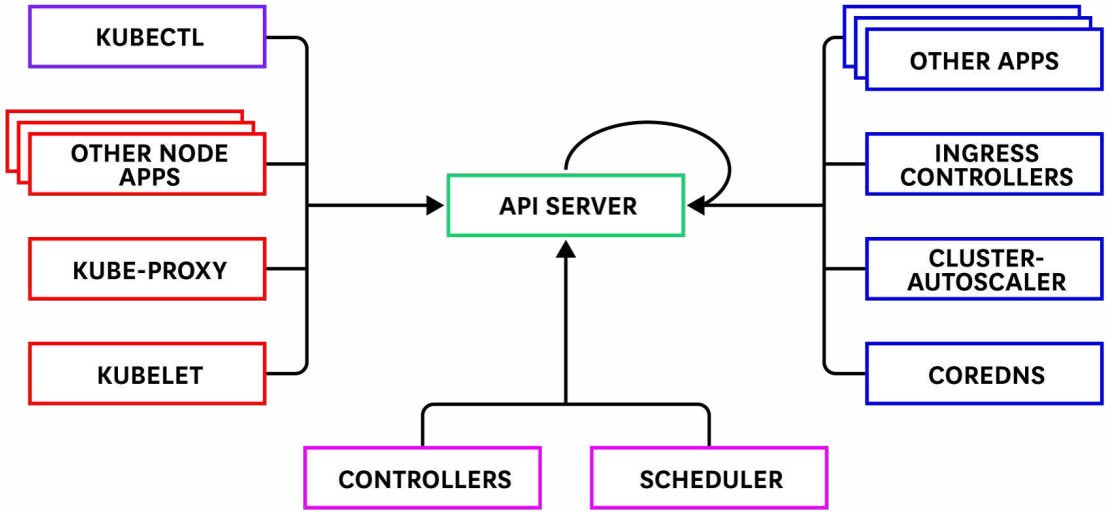


Unavailable pods may also point to capacity issues in your cluster; your nodes may not have enough resource capacity to accept newly launched pods. If your cluster runs on EC2 worker nodes, you can use the Kubernetes Cluster Autoscaler to provision more worker nodes when pods are failing to get scheduled due to resource capacity issues.

CONTROL PLANE MONITORING

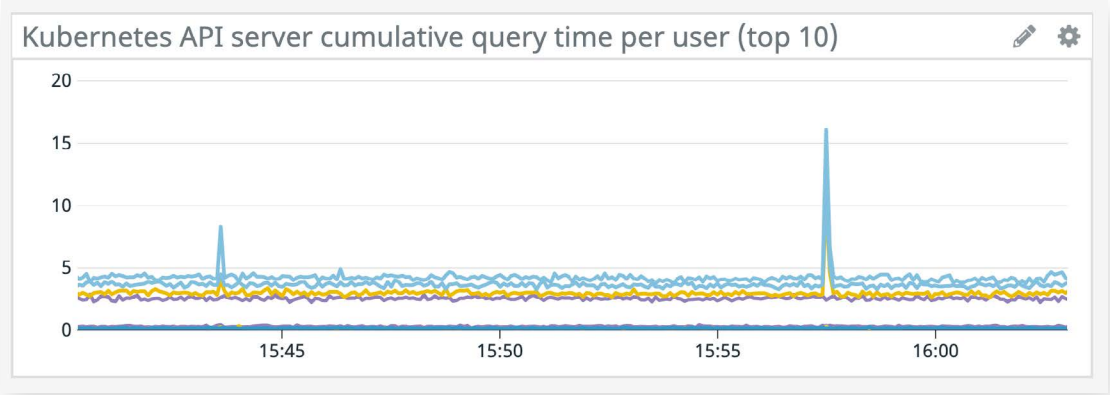
If the control plane cannot communicate efficiently with the rest of the cluster, your applications will not be able to run properly either. In EKS, the control plane is fully managed and abstracted from your view, but you can still monitor control plane metrics to detect cluster-wide problems. These metrics include the latency of requests to the API server and requests to the AWS cloud controller manager, the abstraction layer between Kubernetes and AWS.

For even more granularity, you can also collect control plane audit logs and monitor the latency of specific types of requests to your API server. Because the API server processes all changes to Kubernetes state—and tracks all of this activity in the form of audit logs—these logs can help you retrace critical operations that affect your cluster.



The following control plane data is particularly useful to monitor, whether you're operating a managed service like EKS or a self-hosted Kubernetes cluster.

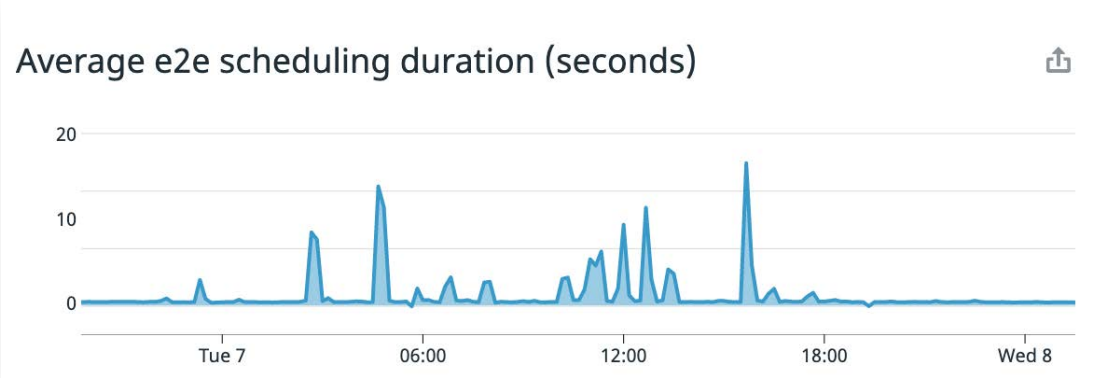
Cumulative API query time per user: This information can be calculated from audit logs and converted into a metric. This metric helps you detect when the API server is responding slowly—and if a specific service or user is overloading the API server with requests. Slow API server response times can ultimately degrade user-facing performance, because all requests—including scaling operations—need to go through the API server. For instance, your application may be loading slowly because overloaded API servers prevented the Horizontal Pod Autoscaler from scaling up pods quickly enough to handle an increase in traffic.



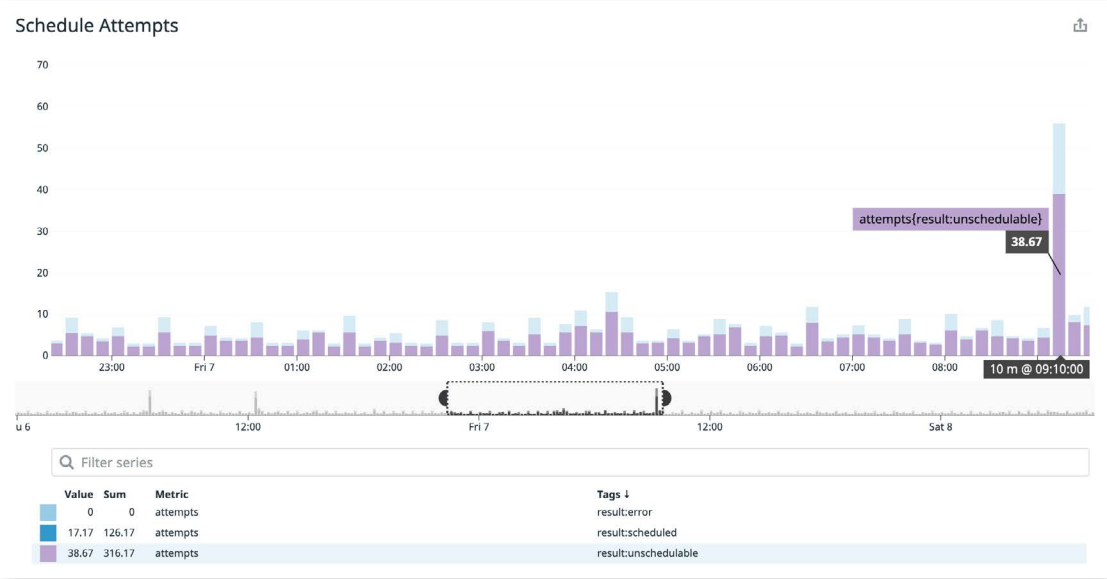
If you see an unexpected spike in this metric, you can inspect where those queries are coming from. This enables you to identify services that are misconfigured and placing unnecessarily high load on the API server. For example, if you use kube2iam and see that it is taking an unusually long time to query the API server (the blue spike in the graph above), you can investigate further by seeing if you recently made any updates to its configuration.

If you accidentally configured each kube2iam pod to query the API server for data about every pod in the cluster, rather than just the pods running on its local node (using the `--node flag`), this will trigger massive load, bringing your API server to a standstill. Keeping an eye on increased API query time can help you detect and remediate such issues before they bring down your cluster.

Scheduling duration: If you see a spike in this metric, it means that pods are taking a long time to get scheduled on nodes, which could ripple into other issues with your applications.



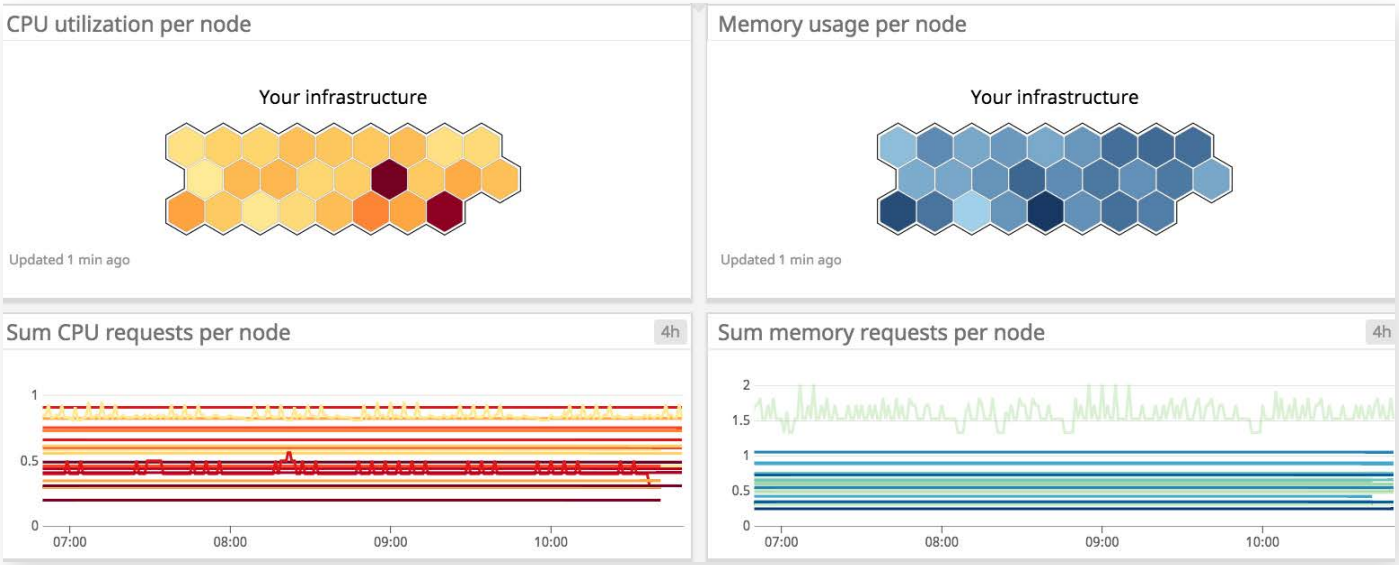
You can also monitor the number of scheduling attempts, broken down by result (error, scheduled, or unschedulable). A large number of unschedulable events could point to a misconfiguration (e.g., overly restrictive taints or tolerations that prevent the scheduler from finding any suitable nodes in your cluster) or may indicate that your nodes do not have enough allocatable resources to satisfy your pods' requirements. Referring to your Kubernetes audit logs can help you determine the reason for unschedulable pods.



MONITORING EKS RESOURCE METRICS

As EKS schedules pods across nodes, they may exhaust all available resources on a node, which can lead to CPU throttling or pod eviction. In your Kubernetes manifests, you have the option to add requests and limits, which specify the amount of CPU (in cores) and memory that each container will use. A request is the minimum amount of CPU or memory that will be allocated to the container, while a limit is the maximum amount that the container will be allowed to use. Pod-level requests and limits are calculated by adding up the requests and limits of the containers that run on a pod.

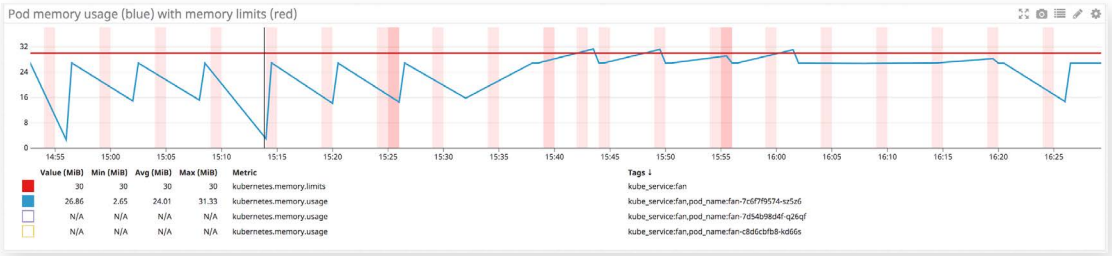
Specifying suitable resource requests and limits can help maintain the right balance between ensuring that your pods have enough available resources to execute their work without over-provisioning (and paying) for resources that won't get used.



The following resource metrics will help you ensure that pods are healthy and able to run smoothly. If you're deploying pods to EC2 infrastructure, you will also need to monitor resource usage on each worker node. Fargate abstracts away this infrastructure layer, meaning that you only need to focus on container- and pod-level resource metrics.

Memory utilization: The kubelet monitors the health of the node it runs on. If the kubelet detects that the node is running low on memory (i.e., if the `MemoryPressure` check emits `True`), it may start evicting pods to free up resources. Under these circumstances, any pod that is using more memory than it has requested is a candidate for eviction.

If you do not specify a memory limit for a pod that runs on an EC2 instance, it could use all available memory on the node. To avoid this scenario, you can specify a memory limit, or the maximum amount of memory a pod is allowed to use. Comparing memory utilization to configured memory limits can help you determine if you need to increase or decrease the limits to meet actual demand. When configuring a memory limit, remember to reserve some memory for system daemons and the kubelet.



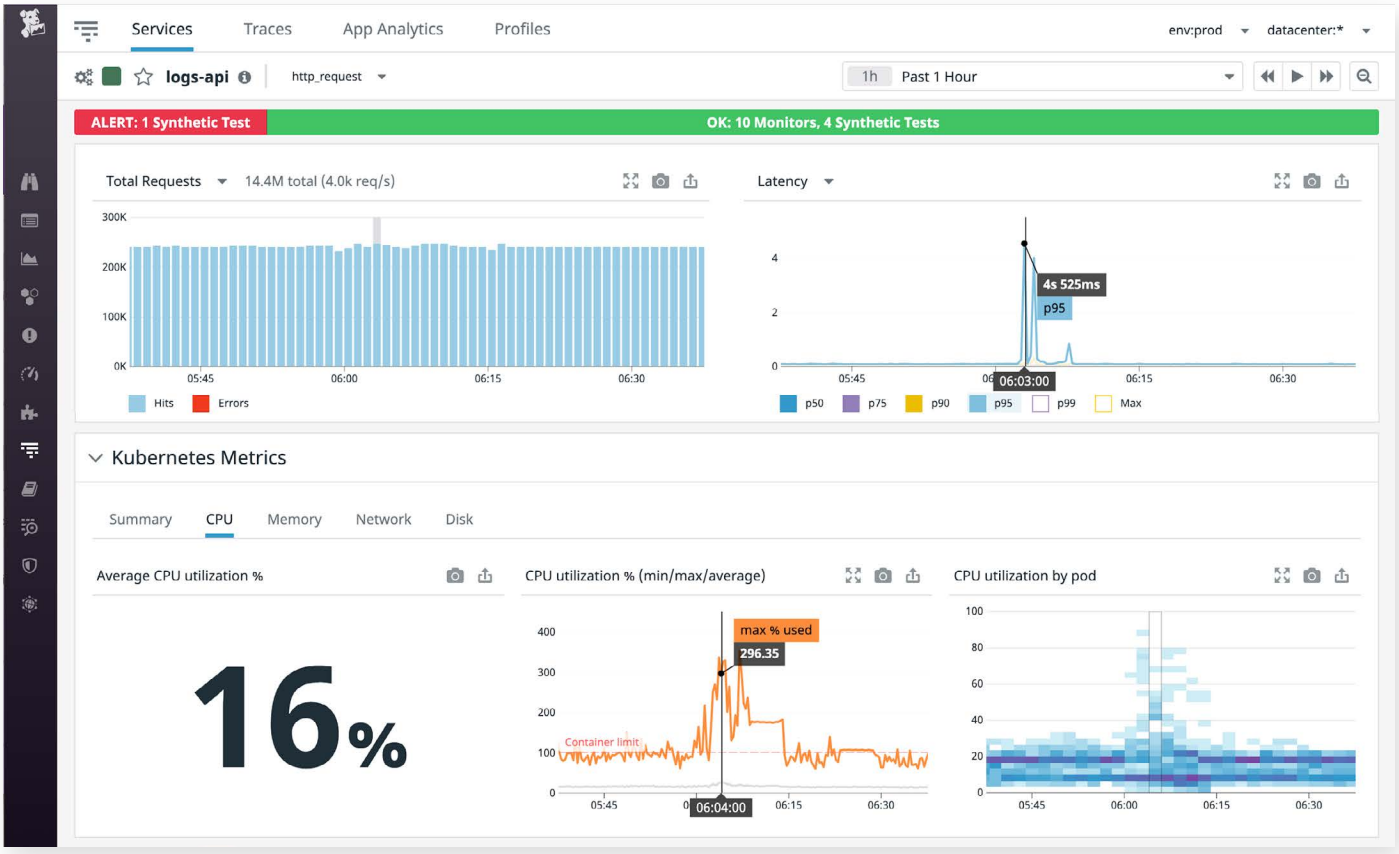
Monitoring resource utilization metrics can also help you avoid setting unnecessarily high resource requirements in your pod specifications. Fargate uses pods' resource requests to determine the amount of compute resources to provision for those pods. If a pod is consistently using less memory than it has requested, you may be able to reduce the request in the EKS pod specification, which would potentially allow Fargate to schedule the workload on a smaller, lower-cost compute resource without negatively affecting performance.

Memory requests per node vs. allocatable memory per node (only EC2): If you are running EKS pods on EC2 nodes, it can be helpful to compare these metrics to determine if your nodes are appropriately sized, and to see if you need to adjust the resource requirements in your pod specifications. The allocatable memory on each node represents the amount of memory that can still be allocated to pods running on that node. Allocatable memory is not equal to the entire capacity of the worker node's instance type—to calculate its actual value, you'll need to subtract the amount of memory reserved by the OS and Kubernetes system processes.

If a node does not have enough allocatable memory to satisfy the minimum memory request for the pod (after accounting for the combined memory requests of the pods it is already running), the pod cannot get scheduled on that node. Keeping an eye on these two metrics, along with desired vs. current pods, can help you detect when your nodes do not have enough capacity to accommodate new pods.

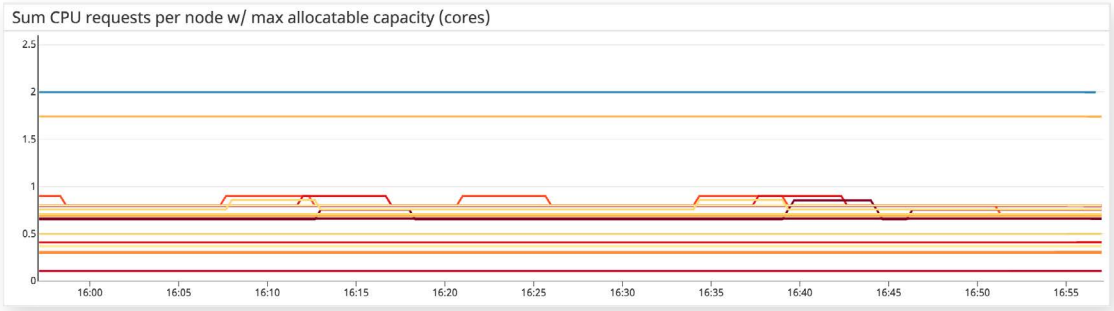
Disk utilization: When a kubelet detects low disk space on its node (e.g., through the `DiskPressure` check), it may start evicting pods to free up resources. Storage is set up differently depending on whether your EKS pods are running on Fargate or EC2. Fargate provides local, ephemeral storage that the containers in each pod can share. For persistent storage, you can provision Amazon Elastic File System (EFS) volumes that your pods can access. If you are using EC2, pods can use EBS volumes—either root volumes on the EC2 instances or persistent volumes (PVs) for persistent storage. Monitoring usage levels on these volumes will help you keep tabs on potential problems before they affect services that rely on them.

CPU utilization: Unlike memory, CPU is a compressible resource, so if a pod exceeds its CPU limit, the kubelet on the node will throttle the amount of CPU available to the pod, but it will not evict it. Correlating high CPU utilization with application performance metrics can help you understand if CPU throttling is contributing to a slowdown in a user-facing service.



Monitoring CPU utilization can also help you determine if you need to adjust your CPU requests and limits or adjust your autoscaling policies to ensure that your cluster can better accommodate your pods' workloads.

CPU requests per node vs. allocatable CPU per node (only EC2): Allocatable CPU measures the amount of CPU resources available to accommodate newly scheduled pods on a node. Kubernetes measures CPU in cores; on EKS clusters, one CPU core is equivalent to one AWS vCPU on an EC2 instance. You can compare the number of requested cores against the number of vCPUs available on your nodes to help inform your capacity planning. Below, you can see the sum of the CPU requests of the pods running on each node compared to the maximum allocatable CPU per node (in blue), based on the instance type.



Network in and network out: EKS sets up networking to ensure that pods can successfully communicate with each other and with the control plane. Monitoring traffic to and from your pods and (if using EC2) nodes can help you determine if there are any networking issues that need to be fixed.

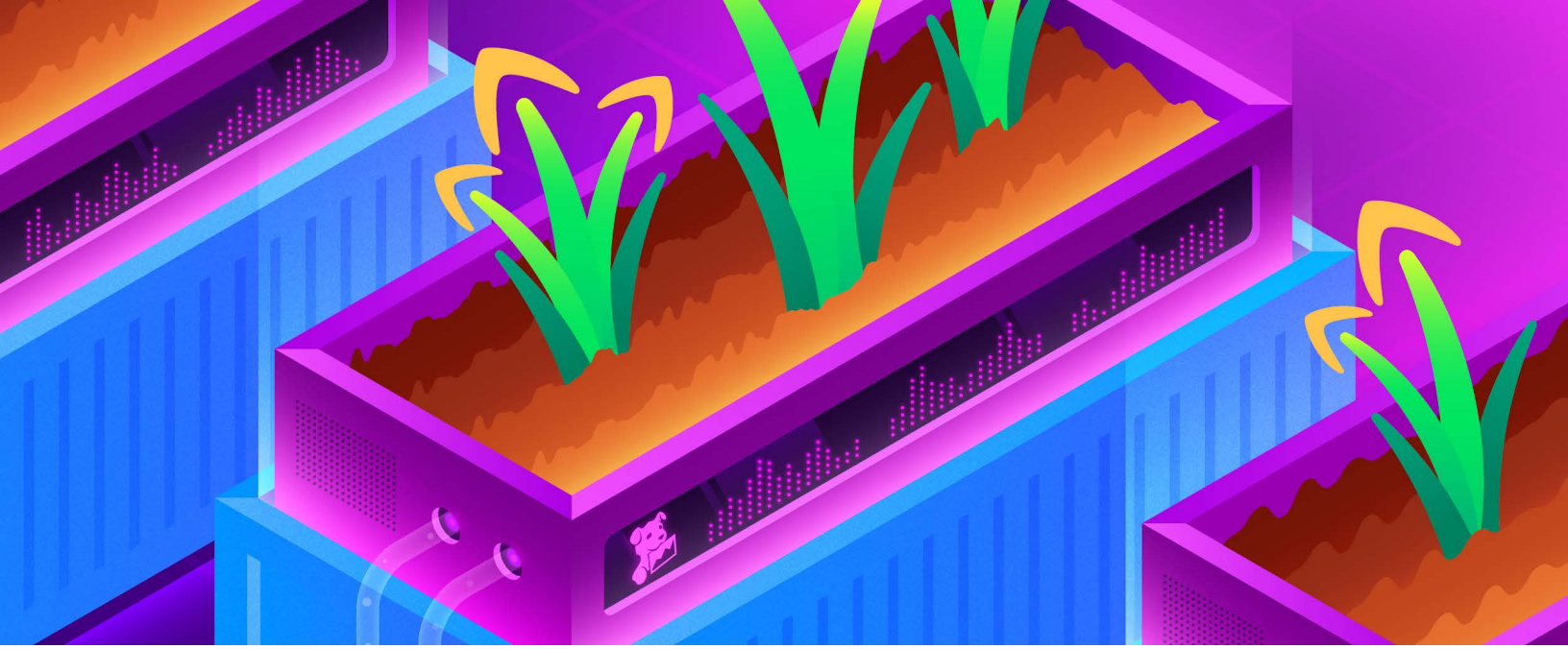
You may also want to correlate network metrics with load balancer metrics to see if network issues may be related to load balancing errors. Note that Fargate currently only supports Application Load Balancers (ALBs), whereas EC2 instances are able to use ALBs as well as Classic Elastic Load Balancers and Network Load Balancers.

Monitoring AWS services that support your orchestrated clusters

Whether you're using a managed orchestration service like Amazon ECS or EKS or self-managing your containers on EC2 instances, your clusters likely depend on other services in the AWS cloud, such as:

- Amazon EC2 for provisioning EKS worker nodes or ECS container instances
- Elastic Load Balancing
- AWS Auto Scaling for dynamically scaling your cluster
- Amazon EBS for providing persistent storage volumes for ECS or EKS workloads running on EC2 instances
- AWS App Mesh for application networking
- AWS CloudFormation for deploying infrastructure as code
- Amazon Elastic Container Registry for managing container images

A comprehensive monitoring solution can provide visibility across every layer of your orchestrated applications and the services they depend on. Next, we will explore how Datadog unifies metrics, traces, and logs from your clusters—and the services they're running—to give you deep insights into your container workloads in one platform.



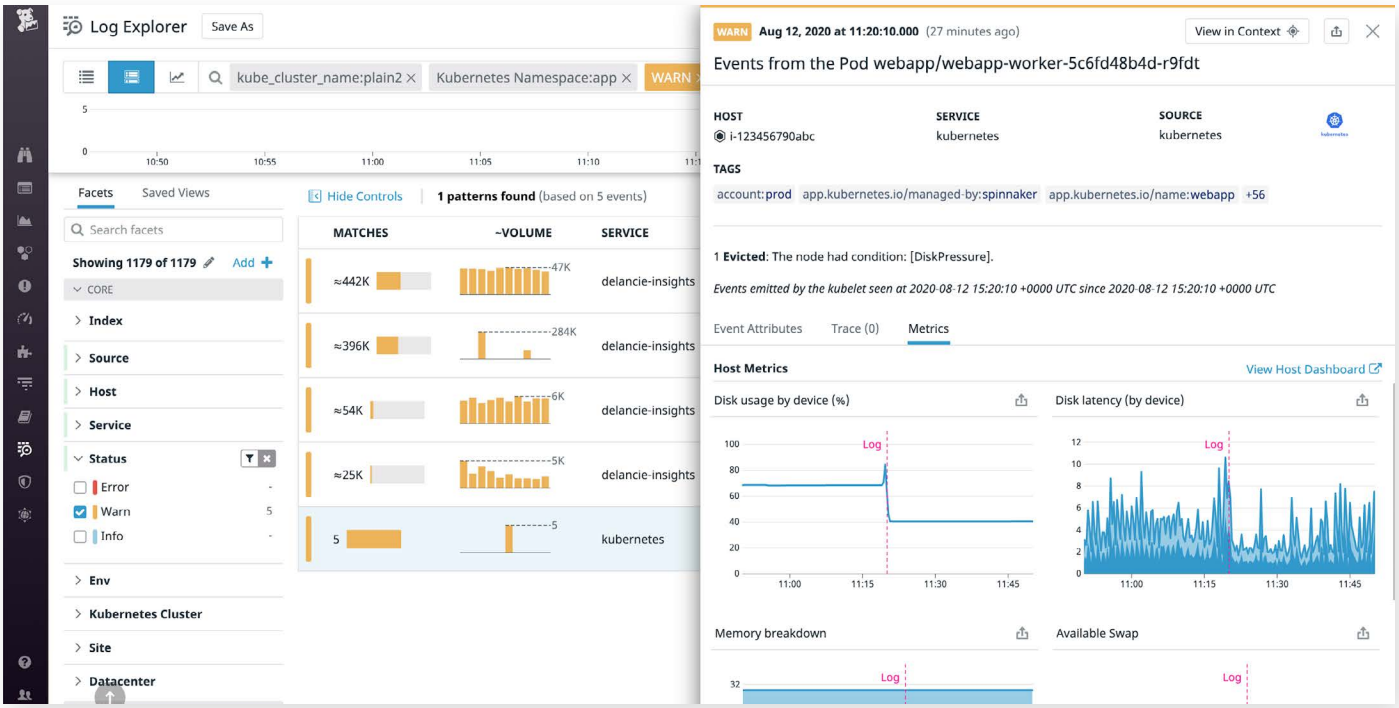
Comprehensive monitoring for AWS container environments

Orchestrated environments are complex and ephemeral, with many layers of abstraction to monitor. For full observability, you need to dynamically track containers as they spin up, shut down, and shift across your infrastructure.

Whether you're running a managed service or hosting Kubernetes clusters on your own EC2 instances, Datadog provides many ways to gather observability data from every component of your orchestrated environment, including:

- Amazon ECS tasks running on EC2 and Fargate
- Amazon EKS pods deployed on EC2 and Fargate
- Kubernetes environments via kube-state-metrics (for cluster state metrics), Metrics Server (for resource usage metrics from containers and nodes), and control plane data
- AWS services that support your orchestrated clusters, via turn-key integrations with Amazon EC2, AWS App Mesh, Elastic Load Balancing, Amazon EBS, and more

In addition to the metrics covered earlier in this section, you can collect distributed traces and logs from the containers, pods, and nodes in your clusters via the open source Datadog Agent. In the Datadog platform, you can easily correlate all of these sources of data to get deeper insights into your applications. For example, the log below shows that the kubelet began reclaiming resources after the node returned `True` for its `DiskPressure` check. Directly from this log, you can click the "Metrics" tab to get more context by viewing correlated resource utilization metrics collected from the node.



The Agent also includes an Autodiscovery feature that automatically configures monitoring for services running in your cluster like Redis and NGINX. You can also configure hundreds of other integrations to gather telemetry data from all the systems, services, and components of your environment. Learn more about these integrations in the documentation.

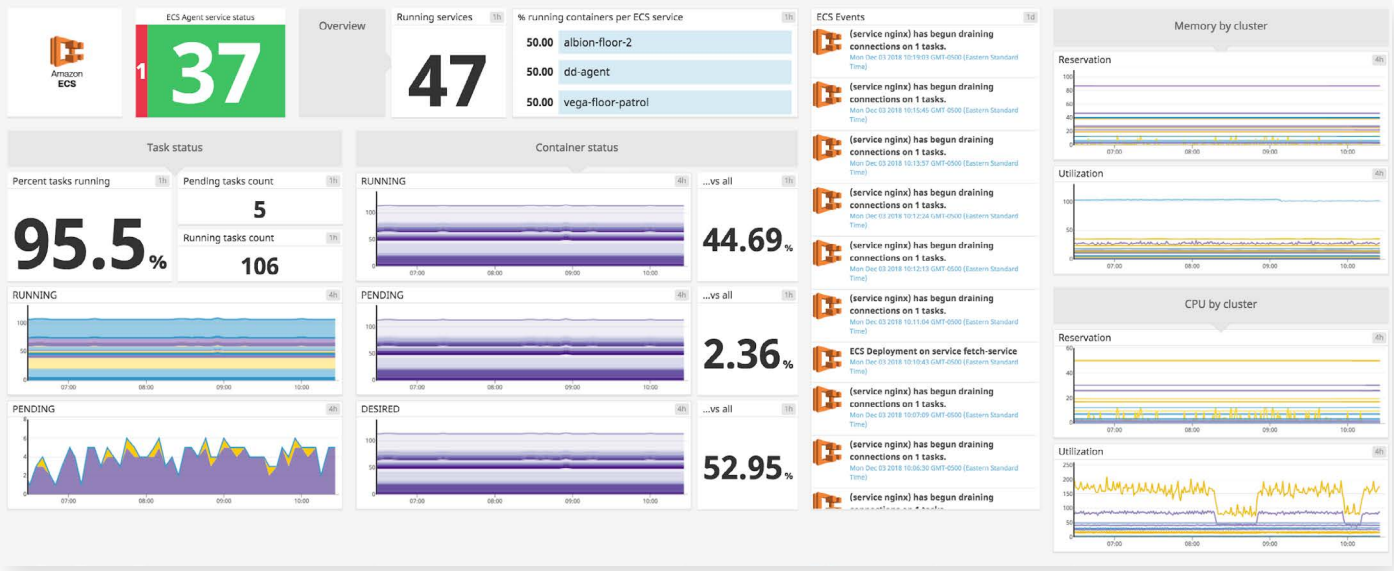
Visualizing your container stack

Once Datadog is collecting real-time monitoring data from your clusters, you can immediately visualize your container applications and infrastructure with:

- out-of-the-box dashboards for Amazon ECS, Kubernetes, and other services
- the Live Container view for granular, process-level insights
- the Container Map for a bird's-eye view of your container infrastructure

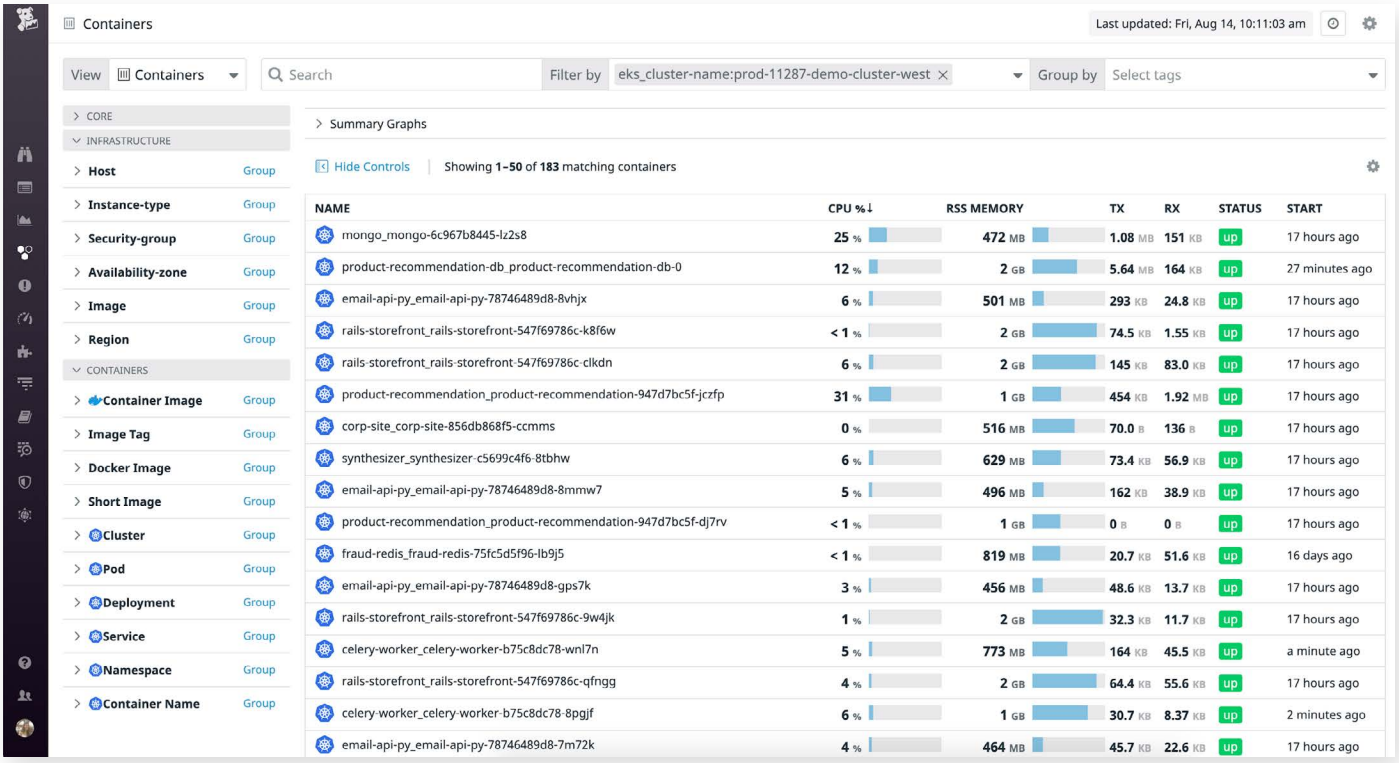
CUSTOMIZABLE DASHBOARDS

Datadog provides out-of-the-box dashboards for Amazon ECS, Kubernetes, and other services so you can quickly visualize key metrics from your environment. The ECS dashboard (shown below) displays the status of tasks and containers in your clusters, as well as service events and resource utilization metrics. To get even deeper visibility into your stack, you can clone and customize any dashboard to include relevant data from load balancers, databases, and other services.



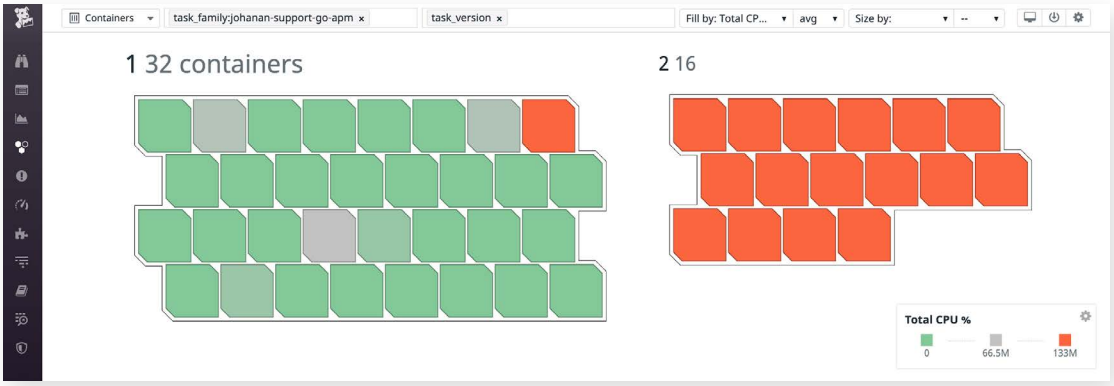
LIVE CONTAINER VIEW

Datadog's Live Container view displays real-time resource consumption metrics from all of your containers. Your telemetry data is automatically tagged with rich metadata that Datadog imports from Amazon CloudWatch, Kubernetes labels, and other sources. With Datadog, you can also implement unified service tagging to ensure that you can correlate and navigate across all your telemetry data with critical tags: `env`, `service`, and `version`. You can use these and any other tags to group and filter the Live Container view across any dimension, such as Kubernetes Deployment or ECS task family, and then drill down to individual containers to debug issues. Because the Datadog Agent collects metrics from your containers at two-second resolution, you'll be able to spot important spikes in resource usage that could be causing issues elsewhere in your application.



CONTAINER MAP

The Datadog Container Map provides a bird's-eye view of your entire container infrastructure. You can select a container-level metric and group and filter by tags to identify hotspots across any dimension of your environment. In the example below, we've used ECS tags to filter for a specific `task_family` and then grouped those containers by `task_version`. This allows us to spot that ECS containers in task version 2 are using a high amount of CPU, compared to containers in task version 1. This indicates that version 2 may include a change that results in increased CPU usage. To continue debugging, the team could take a look at the code changes and determine if any updates are needed.

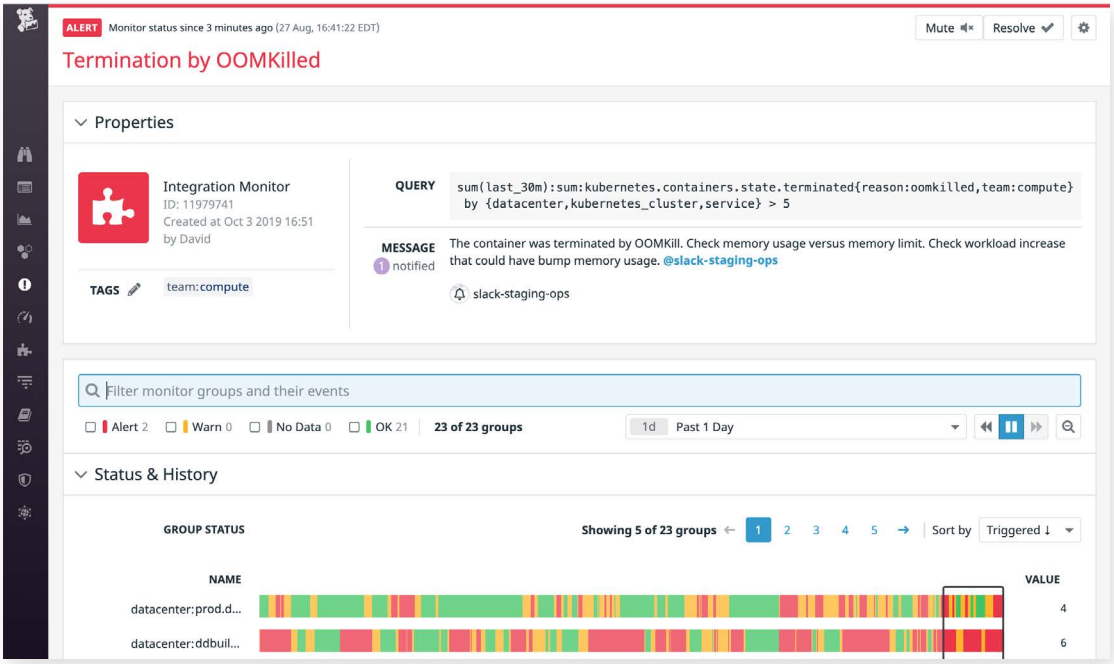


The Container Map can also help you understand if your organization is following best practices. Kubernetes recommends explicitly defining the version number in each container image tag, rather than letting it default to `latest`. This allows you to keep track of the versions running in your environment, so you can revert to a stable version if an update results in a regression or configuration error. Below, grouping the `corp-site` Kubernetes service by `image_tag` allows us to see how many containers are using the `latest` version as opposed to a specific image digest, as recommended.

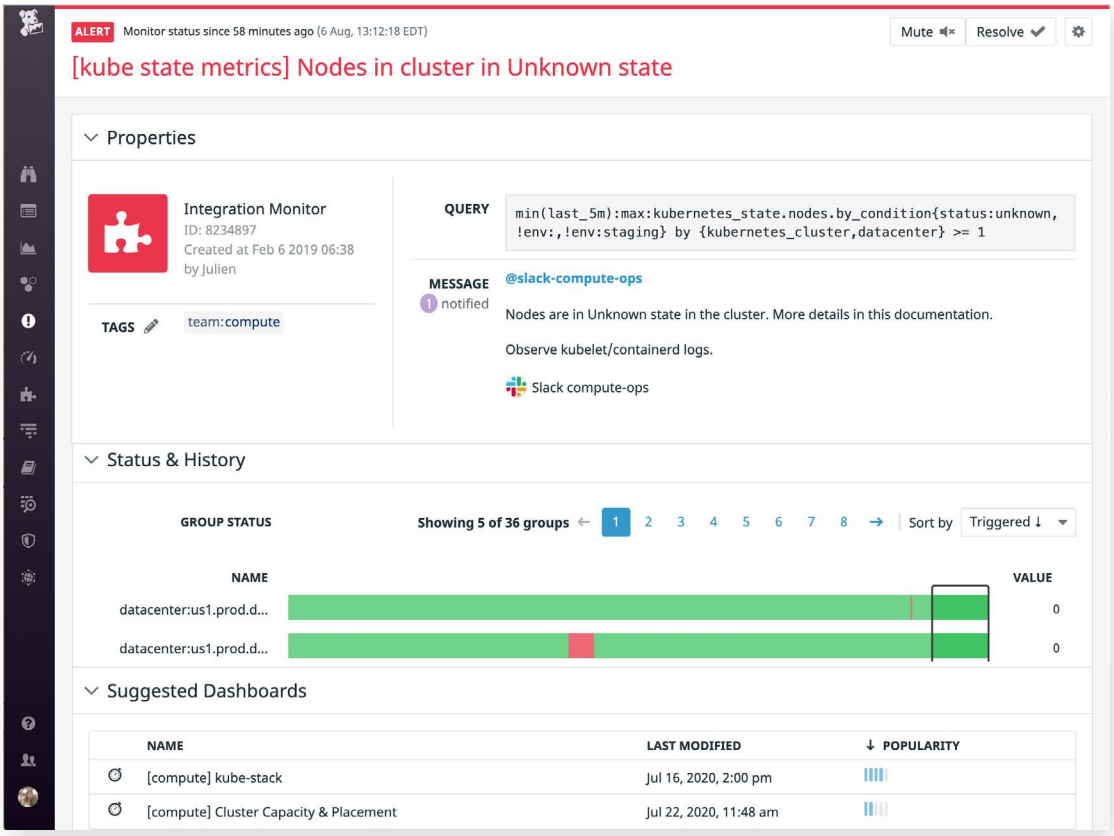
Automated alerting

Monitors can help you automatically track issues, ranging from high application latency to resource capacity issues like out-of-memory errors. Below, we'll walk through a few examples of alerts that can immediately surface potential issues in your orchestrated clusters.

With Datadog, you can create monitors to automatically notify your team about important issues such as terminated pods, high disk usage on Kubernetes nodes, pods in `CrashLoopBackoff` status, and more. The example below configures a monitor to detect when several containers were terminated for hitting their memory limits.



The monitor below will send a notification to a team-based Slack channel if Datadog detects that at least one node has remained in `Unknown` state for a certain period of time. This monitor excludes data from staging, since our Kubernetes telemetry data is tagged by environment.



When the node controller fails to receive a response from a node over a certain period of time (40 seconds, by default), it marks the condition of that node as `Unknown`. If a node remains in `Unknown` state over a specific duration (the length of your configured `pod-eviction-timeout`), the node controller will start deleting pods on that node. By configuring Datadog to alert you about nodes that are stuck in `Unknown` state, you can detect and fix this issue before pods get deleted.

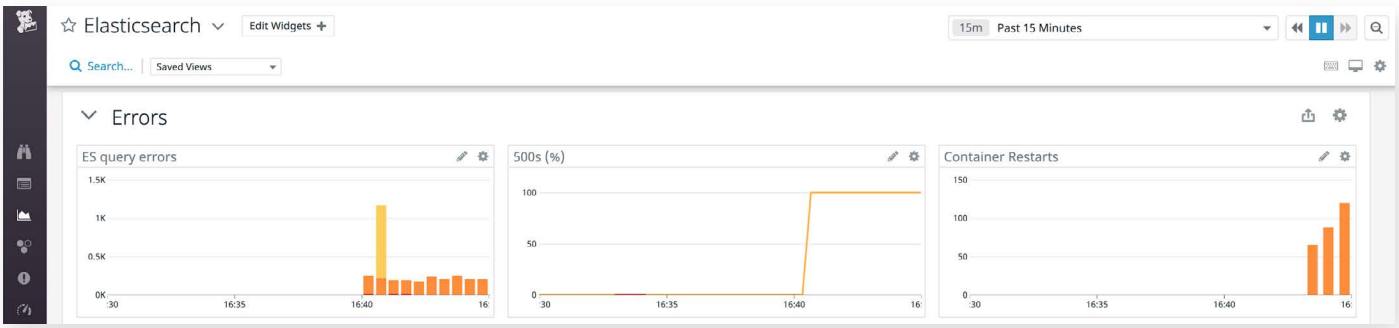
Alerts are a critical aspect of comprehensive monitoring, but they are just one piece of the puzzle. When you get alerted, you need to be able to quickly investigate the issue. Datadog helps facilitate your incident response process by helping you easily navigate across correlated metrics, distributed traces, and logs from your containerized applications, so you can get all the context you need for root cause analysis.

Holistic monitoring and incident response

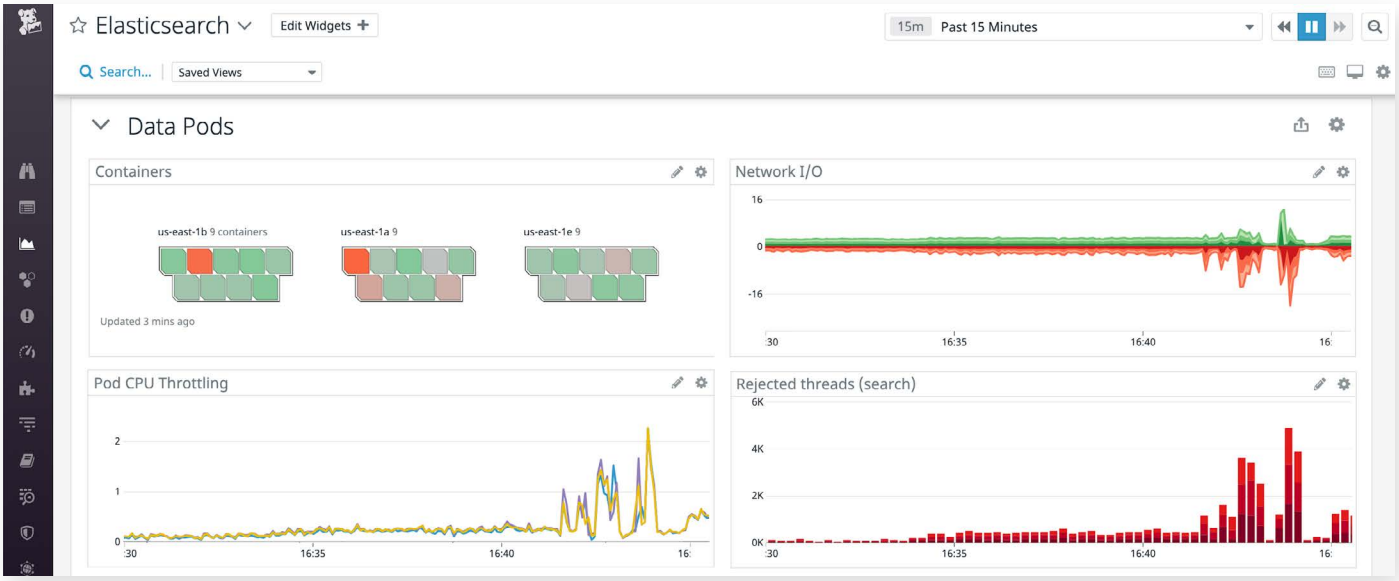
Failures and outages are unavoidable when you're running complex, distributed systems. In dynamic, ephemeral environments, it simply isn't feasible to alert on every resource capacity issue or scheduling error—especially because Kubernetes is designed to be self-healing, meaning that many issues are likely to resolve on their own.

To reduce alert fatigue, we recommend alerting on symptoms of issues, rather than the underlying causes, which may not always directly affect your users. This means, for instance, alerting on user-facing errors or performance issues (e.g., very slow load times or an inaccessible application endpoint) instead of resource issues that don't necessarily translate into a poor user experience (e.g., high CPU usage). Within your alert notifications, you can include links to dashboards that serve as useful resources for investigation.

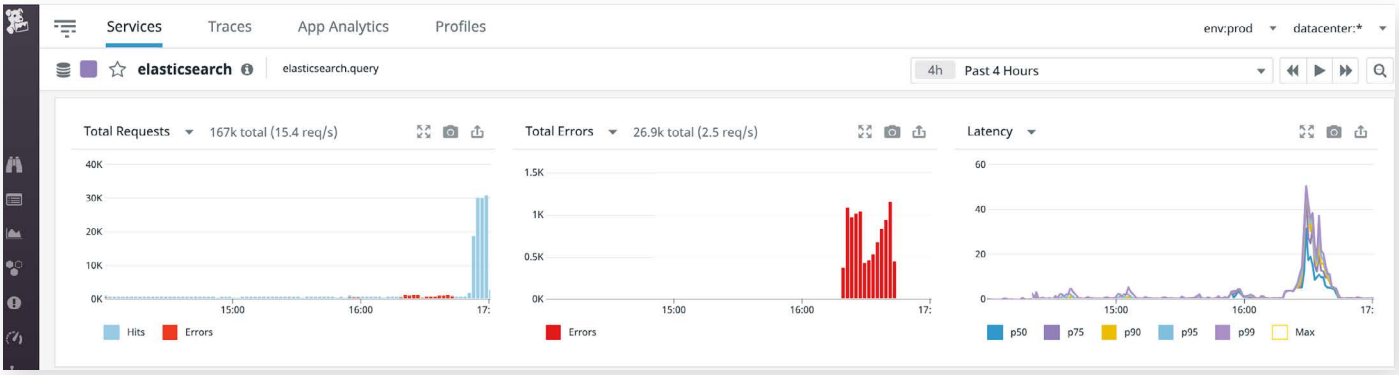
Let's walk through an example of how we would investigate an availability issue in a Kubernetes-orchestrated environment. We just received an alert of a high rate of 500 errors on a user-facing endpoint—a critical indication that something is wrong. This endpoint relies on Elasticsearch to query and display data to users. The alert notification included a link to a dashboard that we pre-built for this service, allowing us to quickly get a sense of what's going on. In the dashboard, we can see that a high rate of Elasticsearch query errors occurred around the same time as 500 errors on the service.



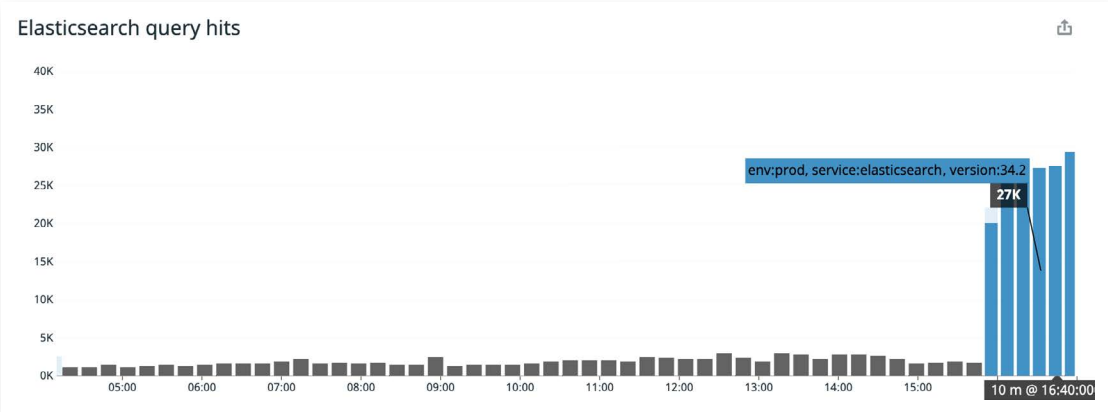
Container-level resource metrics indicate that the Elasticsearch data pods exhibited increased CPU throttling and search thread pool rejections. Elasticsearch uses thread pools to manage its workloads; when it receives more requests than it is able to process, it adds them to the thread pool queue. In this case, Elasticsearch began rejecting requests after the search thread pool queue reached its maximum size.



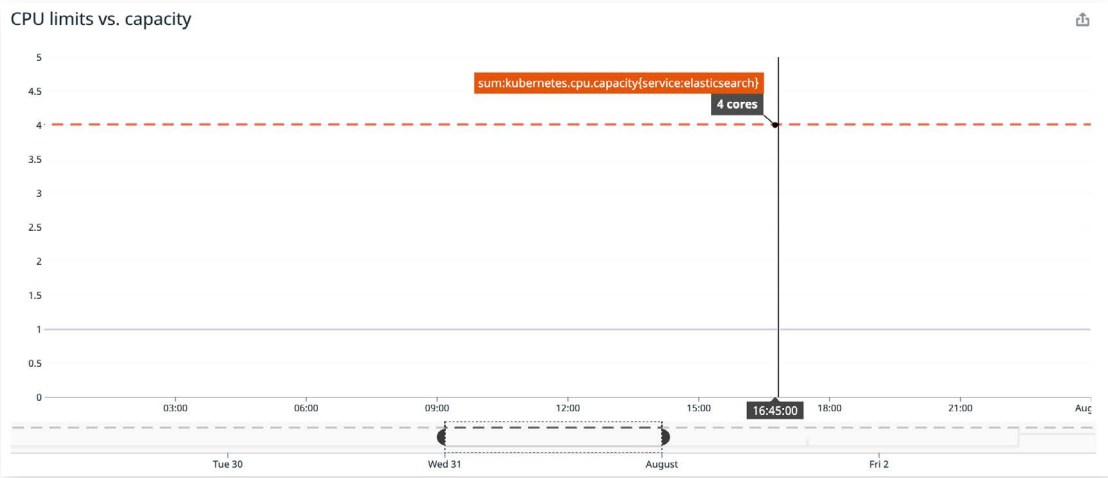
To get more context around the scope of the issue, we decide to take a look at the service-level dashboard in Datadog APM. From this dashboard, we can see that this service recently started receiving an unusually high number of queries.



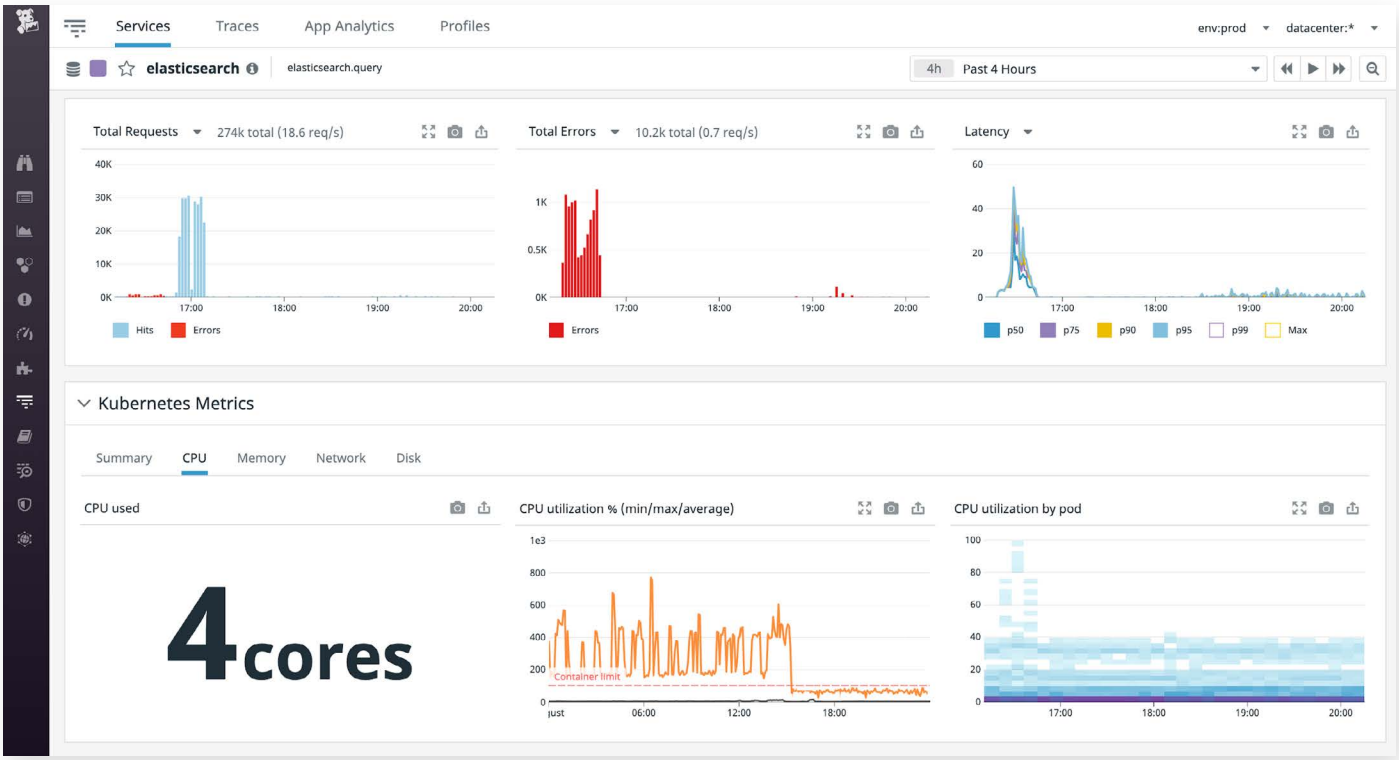
If we break down the request throughput metric by a `version` tag, we can clearly see that query throughput increased after the latest release (in blue) compared to the previous one (in gray). To investigate further, we inspect the application code that was updated in that version. When we do, we find a bug: we are retrying search queries too aggressively—up to six times without any backoff logic. This helps explain the increased rate of requests we saw in the Datadog APM dashboard.



As part of our investigation, we also decide to take a closer look at the resource requests and limits configured in this cluster. We discover that the CPU limit is too low—much lower than the CPU capacity of the EC2 instance type we are using. Kubernetes was throttling CPU on containers whenever they hit their unnecessarily low CPU limits, even though the node was not yet using its full CPU capacity.



Since we configure Kubernetes to schedule only one Elasticsearch pod on each node in this cluster, we decide to raise the CPU limits (while still reserving some CPU for the system daemons and the kubelet to run) so that containers can leverage more of the CPU capacity of the EC2 instances they are running on. Once we configure a higher CPU limit and fix the bug in the retry logic, the service recovers. Because pod-level CPU utilization and CPU limit metrics are automatically included in this service-level dashboard, we can easily spot if CPU throttling correlates with application performance issues in the future.



Full-stack observability for dynamic environments

In this eBook, we've explored how managed orchestration services like Amazon ECS and Amazon EKS help you deploy container workloads at scale—and the key metrics to monitor when using each of these services. We've also seen how Fargate provides the option to run your ECS or EKS containers on a serverless compute engine, reducing the need to manage your own infrastructure.

Whether you are using a managed service or operating Kubernetes on your own EC2 instances, full-stack observability is crucial for troubleshooting user-facing issues. We've seen how Datadog aggregates telemetry data from every layer of your stack so you can effectively investigate potential root causes of critical issues, ranging from bad code deploys to misconfigured Kubernetes resource limits.

Further reading

To learn more about monitoring containerized applications, refer to the following resources:

- Amazon ECS monitoring guide
- Amazon EKS monitoring guide
- Kubernetes audit logs monitoring guide

If you would like to put these monitoring strategies in action for your environment, you can sign up for a full-featured Datadog trial at www.datadog.com.


aws
DATADOG