

THE STATE OF THE KUBERNETES ECOSYSTEM

The New Stack

The State of the Kubernetes Ecosystem

Alex Williams, Founder & Editor-in-Chief

Core Team:

Bailey Math, AV Engineer

Benjamin Ball, Marketing Director

Gabriel H. Dinh, Executive Producer

Judy Williams, Copy Editor

Kiran Oliver, Associate Podcast Producer

Krishnan Subramanian, Technical Editor

Lawrence Hecht, Research Director

Scott M. Fulton III, Editor & Producer

TABLE OF CONTENTS

Introduction 4

Sponsors 8

THE STATE OF THE KUBERNETES ECOSYSTEM

An Overview of Kubernetes and Orchestration..... 9

Google Cloud: Plotting the Kubernetes Roadmap 35

CNCF: Kubernetes 1.7 and Extensibility..... 36

Map of the Kubernetes Ecosystem 37

Codeship: Orchestration and the Developer Culture..... 46

User Experience Survey..... 47

Twistlock: Rethinking the Developer Pipeline..... 92

Buyer’s Checklist to Kubernetes..... 93

Red Hat OpenStack: Cloud-Native Apps Lead to Enterprise Integration..... 113

Issues and Challenges with Using Kubernetes in Production..... 114

CoreOS: Maintaining the Kubernetes Life Cycle..... 141

Roadmap for the Future of Kubernetes 142

Closing..... 172

KUBERNETES SOLUTIONS DIRECTORY

Kubernetes Distributions..... 175

Tools and Services 180

Relevant DevOps Technologies..... 184

Relevant Infrastructure Technologies 188

Disclosures..... 191

INTRODUCTION

The most fundamental conception is, as it seems to me, the whole system, in the sense of physics, including not only the organism-complex, but also the whole complex of physical factors forming what we call the environment. ... Though the organism may claim our primary interest, when we are trying to think fundamentally, we cannot separate them from their special environment, with which they form one physical system. It is the systems so formed from which, from the point of view of the ecologist, are the basic units of nature on the face of the earth. These are ecosystems.

-Sir Arthur Tansley, "The Use and Abuse of Vegetational Concepts and Terms," 1935.

We use the term infrastructure more and more to refer to the support system for information technology. Whatever we do with our applications that creates value for our customers, or generates revenue for ourselves, we're supporting it now with IT infrastructure. It's all the stuff under the hood. It's also the part of technology that, when it works right or as well as we expect, we don't stand in long lines to get a glimpse of, nor do we see much discussion of it on the evening news.

In the stack of technologies with which we work today, there is a growing multitude of layers that are under the hood. With modern hyperconverged servers that pool their compute, storage and memory resources into colossal pools, the network of heterogeneous technologies with which those pools are composed is one layer of physical infrastructure.

And in a modern distributed computing network, where even the cloud can be only partly in the cloud, the support structure that makes applications deployable, manageable, and scalable has become our virtual infrastructure. Yes, it's still under the hood, only it's the hood at the very top of the stack.

This book is about one very new approach to virtual infrastructure — one that emerged as a result of Google's need to run cloud-native applications on a massively scaled network. Kubernetes is not really an operating system, the way we used to think of Windows Server or the many enterprise flavors of Linux. But in a growing number of organizations, it has replaced the operating system in the minds of operators and developers. It is a provider of resources for applications designed to run in containers (what we used to call "Linux containers," though whose form and format have extended beyond Linux), and it ensures that the performance of those applications meets specified service levels. So Kubernetes does, in that vein, replace the operating system.

The title of this book refers to the Kubernetes ecosystem. This is an unusual thing to have to define. The first software ecosystems were made up of programmers, educators and distributors who could mutually benefit from each other's work. Essentially, that's what the Kubernetes ecosystem tries to be. It foresees an environment whose participants leverage the open source process, and the ethics attached to it, to build an economic system whose participants all benefit from each other's presence.

Only it's hard to say whether Kubernetes actually is, or should be, at the center of this ecosystem. Linux is no longer the focal point of the ecosystem to which Linux itself gave rise. A distributed computing environment is composed of dozens of components — some of them open source, some commercial, but many of them both. Kubernetes may

have given rise to one scenario where these components work in concert, but even then, it's just one component. And in a market where ideas are thriving once again with far less fear of patent infringement, that component may be substituted.

The purpose of this book is to give you a balance of comprehension with conciseness, in presenting for you the clearest snapshot we can of the economic and technological environment for distributed systems, and Kubernetes' place in that environment. We present this book to you with the help and guidance of six sponsors, for which we are grateful:

- [**Cloud Native Computing Foundation \(CNCF\)**](#), a Linux Foundation project; the steward of the Kubernetes open source project and its many special interest groups; and also the steward of Fluentd, linkerd, Prometheus, OpenTracing, gRPC, CoreDNS, containerd, rkt and CNI.
- [**Codeship**](#), a continuous integration platform provider that integrates Docker and Kubernetes.
- [**CoreOS**](#), producer of the Tectonic commercial platform, which incorporates upstream Kubernetes as its orchestration engine, alongside enterprise-grade features.
- Powered by Kubernetes, Google's Container Engine on [**Google Cloud Platform**](#) is a managed environment used to deploy containerized applications.
- [**Red Hat**](#), producer of the OpenShift cloud-native applications platform, which utilizes Kubernetes as its orchestration engine.
- [**Twistlock**](#), which produces an automated container security platform designed to be integrated with Kubernetes.

Portions of this book were produced with contributions from software engineers at

- [Kenzan](#), a professional services company that crafts custom IT deployment and management solutions for enterprises.

We're happy to have you aboard for this first in our three-volume series on Kubernetes and the changes it has already made to the way businesses are deploying, managing and scaling enterprise applications.

SPONSORS

We are grateful for the support of our ebook foundation sponsor:

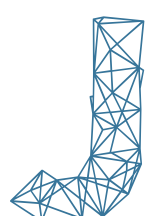


And our sponsors for this ebook:



AN OVERVIEW OF KUBERNETES AND ORCHESTRATION

by **JANAKIRAM MSV** and **KRISHNAN SUBRAMANIAN**

ust a few years ago, the most likely place you'd expect to find a functional Linux container — whether it be the old **cgroup** style, or a full-blown Docker or CNCF [rkt](#) container — was in an isolated, sandbox environment on some developer's laptop. Usually, it was an experiment. At best, it was a workbench. But it wasn't part of the data center.

Today, containers have emerged as the de facto choice for deploying new, cloud-native applications in production environments. Within a three- to four-year span of time, the face of modern application deployment has transformed from virtual machine-based cloud platforms, to orchestrated containers at scale.

In this chapter, we will discuss the role orchestrators (including Kubernetes) play in the container ecosystem, introduce some of the major orchestration tools in the market, and explain their various benefits.

How Kubernetes Got Here

The idea of containerization is not new. Some form of virtual isolation,

whether for security or multi-tenancy purposes, has been bandied about the data center since the 1970s.

Beginning with the advent of the `chroot` system call, first in Unix and later in BSD, the idea of containerization has been part of enterprise IT folklore. From FreeBSD Jails to Solaris Zones to Warden to LXC, containers have been continuously evolving, all the while inching closer and closer to mainstream adoption.

Well before containers became popular among developers, Google was running some of its core web services in Linux containers. In a [presentation](#) at GlueCon 2014, Joe Beda, one of Kubernetes' creators, claimed that Google launches over two billion containers in a week. The secret to Google's ability to manage containers at that scale lies with its internal data center management tool: [Borg](#).

Google redeveloped Borg into a general-purpose container orchestrator, later releasing it into open source in 2014, and donating it to the Cloud Native Computing Foundation (CNCF) project of the Linux Foundation in 2015. Red Hat, CoreOS, Microsoft, ZTE, Mirantis, Huawei, Fujitsu, Weaveworks, IBM, Engine Yard, and SOFTICOM are among the key contributors to the project.

After Docker arrived in 2013, the adoption level of containers exploded, catapulting them into the spotlight for enterprises wanting to modernize their IT infrastructure. There are four major reasons for this sudden trend:

- **Encapsulation:** Docker solved the user experience problem for containers by making it easier for them to package their applications. Before Docker, it was painfully difficult to handle containers (with the exception of Warden, which was abstracted out by the Cloud Foundry platform).

- **Distribution:** Ever since the advent of cloud computing, modern application architectures have evolved to become more distributed. Both startups and larger organizations, inspired by the emerging methodologies and work ethics of DevOps, have in recent years turned their attentions to microservices architecture. Containers, which are by design more modular, are better suited for enabling microservices than any other architecture to date.
- **Portability:** Developers love the idea of building an app and running it anywhere — of pushing the code from their laptops to production, and finding they work in exactly the same way without major modifications. As Docker accumulated a wider range of tools, the breadth and depth of functionality helped spur developers' adoption of containers.
- **Acceleration:** Although forms of containerization did exist prior to Docker, their initial implementations suffered from painfully slow startup times — in the case of LXC, several minutes. Docker reduced that time to mere seconds.

Since its initial release in July 2015, Kubernetes has grown to become the most popular container orchestration engine. Three of the top four public cloud providers — Google, IBM and Microsoft — offered Containers as a Service (CaaS) platforms based on Kubernetes at the time of this publication. The fourth, Amazon, just joined the CNCF with its own plans to support the platform. Although Amazon does have its own managed container platform in the form of [EC2 Container Service](#), AWS is known for running the most Kubernetes clusters in production. Large enterprises such as education publisher Pearson, the Internet of Things appliance division of Philips, TicketMaster, eBay and The New York Times Company are running Kubernetes in production.

What is Orchestration?

While containers helped increase developer productivity, orchestration tools offer many benefits to organizations seeking to optimize their DevOps and Ops investments. Some of the benefits of container orchestration include:

- Efficient resource management.
- Seamless scaling of services.
- High availability.
- Low operational overhead at scale.
- A declarative model (for most orchestration tools) reducing friction for more autonomous management.
- Operations-style Infrastructure as a Service (IaaS), but manageable like Platform as a Service (PaaS).

Containers solved the developer productivity problem, making the DevOps workflow seamless. Developers could create a Docker image, run a Docker container and develop code in that container. Yet this introduction of seamlessness to developer productivity does not translate automatically into efficiencies in production environments.

Quite a bit more separates a production environment from the local environment of a developer's laptop than mere scale. Whether you're running n-tier applications at scale or microservices-based applications, managing a large number of containers and the cluster of nodes supporting them is no easy task. Orchestration is the component required to achieve scale, because scale requires automation.

The distributed nature of cloud computing brought with it a paradigm shift

in how we perceive virtual machine infrastructure. The notion of “cattle vs. pets” — treating a container more as a unit of livestock than a favorite animal — helped reshape people’s mindsets about the nature of infrastructure. Putting this notion into practice, containers at scale extended and refined the concepts of scaling and resource availability.

The baseline features of a typical container orchestration platform include:

- Scheduling.
- Resource management.
- Service discovery.
- Health checks.
- Autoscaling.
- Updates and upgrades.

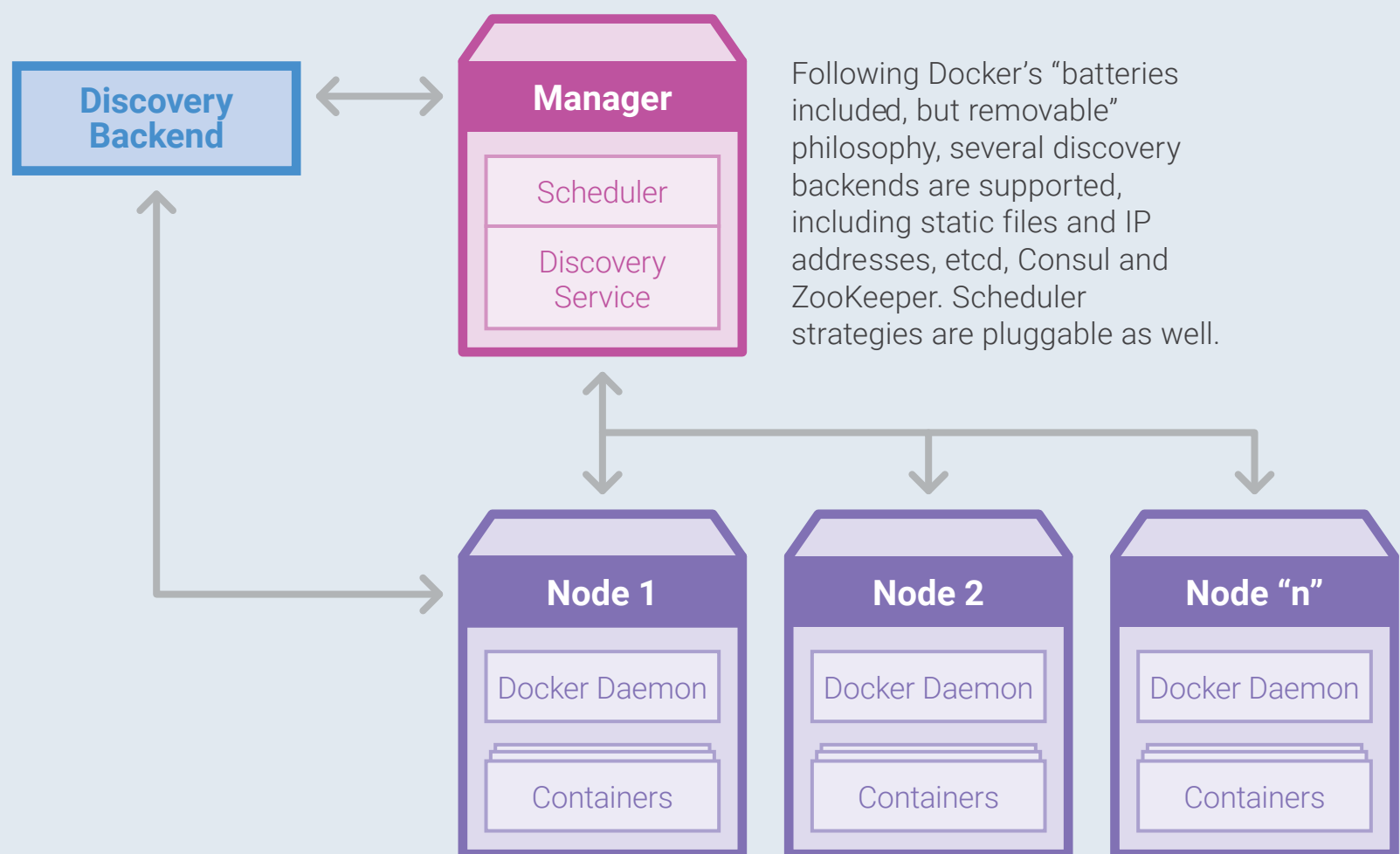
The container orchestration market is currently dominated by open source software. At the time of this publication, Kubernetes leads the charge in this department. But before we dig deeper into Kubernetes, we should take a moment to compare it to some of the other major orchestration tools in the market.

Docker Swarm

Docker, Inc., the company responsible for the most popular container format, offers Docker Swarm as its orchestration tool for containers.

With Docker, all containers are standardized. The execution of each container at the operating system level is handled by runc, an implementation of the Open Container Initiative (OCI) specification. Docker works in conjunction with another open source component, containerd, to manage the life cycle of containers executed on a specific

Docker Swarm: Swap, Plug, and Play



Source: The New Stack

THE NEW STACK

FIG 1.1: *The relationship between master and worker nodes in a typical Docker Swarm configuration.*

host by runc. Together, Docker, containerd, and the runc executor handle the container operations on a host operating system.

Simply put, a swarm — which is orchestrated by Docker Swarm — is a group of nodes that run Docker. Such a group is depicted in Figure 1.1. One of the nodes in the swarm acts as the manager for the other nodes, and includes containers for the scheduler and service discovery component.

Docker's philosophy requires standardization at the container level and uses the Docker application programming interface (API) to handle orchestration, including the provisioning of underlying infrastructure. In keeping with its philosophy of "batteries included but removable," Docker Swarm uses the existing Docker API and networking framework without extending them, and integrates more nicely with the Docker Compose

tool for building multi-container applications. It makes it easier for developers and operators to scale an application from five or six containers, to hundreds.

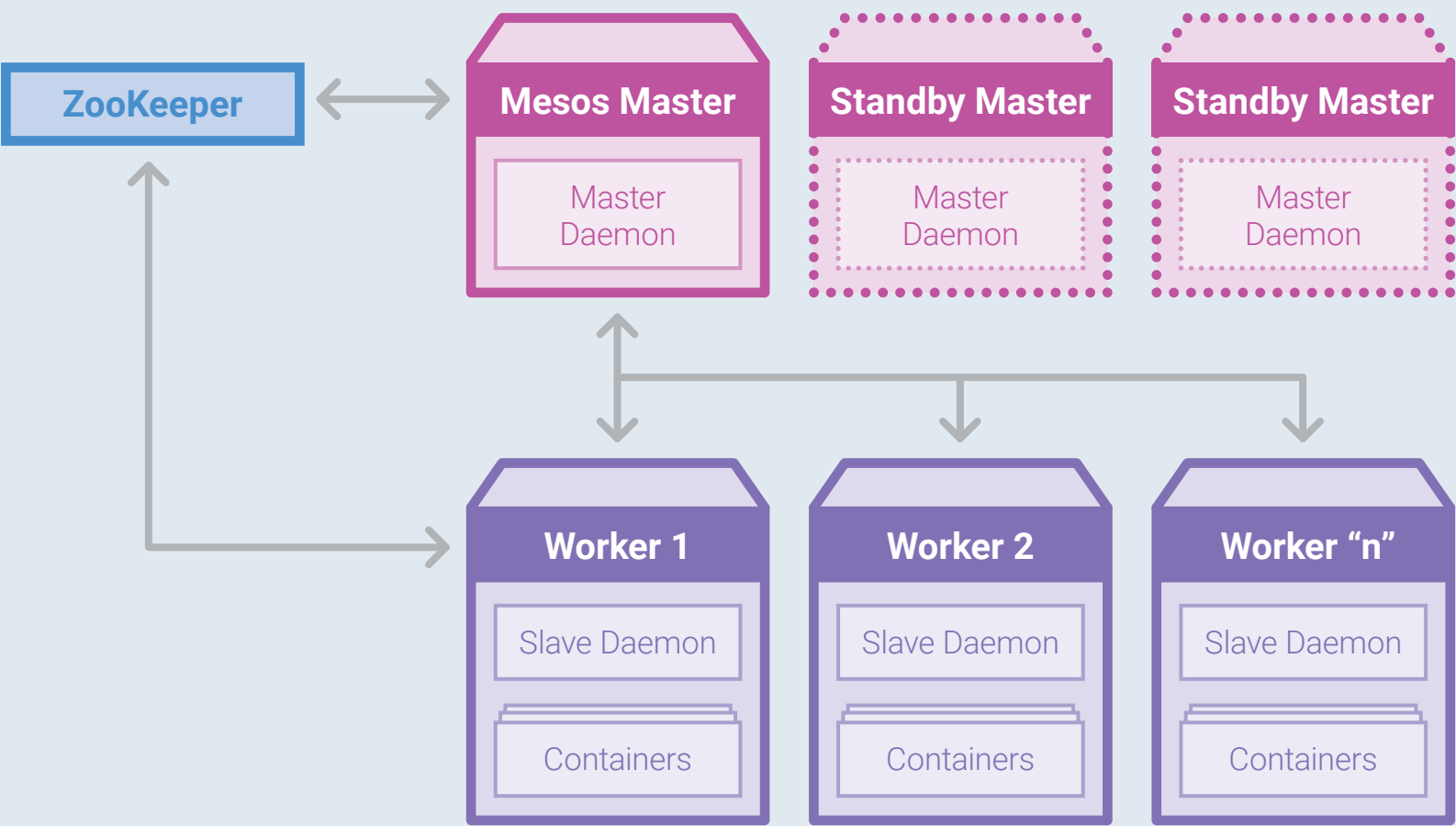
NOTE: Docker Swarm uses the Docker API, making it fit easily into existing container environments. Adopting Docker Swarm may mean an all-in bet on Docker, Inc. Currently, Swarm’s scheduler options are limited.

Apache Mesos

Apache Mesos is an open source cluster manager that pre-dates Docker Swarm and Kubernetes. Coupled with Marathon, a framework for orchestrating container-based applications, it offers an effective alternative to Docker Swarm and Kubernetes. Mesos may also use other frameworks to simultaneously support containerized and non-containerized workloads.

FIG 1.2: *Apache Mesos, built for multifarious, high-performance workloads.*

Apache Mesos: Built for High-Performance Workloads



Mesos' platform, depicted in Figure 1.2, shows the master/worker relationship between nodes. In this scheme, distributed applications are coordinated across a cluster by a component called the ZooKeeper. It's the job of this ZooKeeper to elect masters for a cluster, perhaps apportion standby masters, and instill each of the other nodes with agents. These agents establish the master/worker relationship. Within the master, the master daemon establishes what's called a "framework" that stretches, like a bridge, between the master and worker nodes. A scheduler running on this framework determines which of these workers is available for accepting resources, while the master daemon sets up the resources to be shared. It's a complex scheme, but it has the virtue of being adaptable to many types and formats of distributed payload — not just containers.

Unlike Docker Swarm, Mesos and Marathon each has its own API, making the two of them much more complex to set up together, compared with other orchestration tools. However, Mesos is much more versatile in supporting Docker containers alongside hypervisor-driven virtual machines such as VMware vSphere and KVM. Mesos also enables frameworks for supporting big data and high-performance workloads.

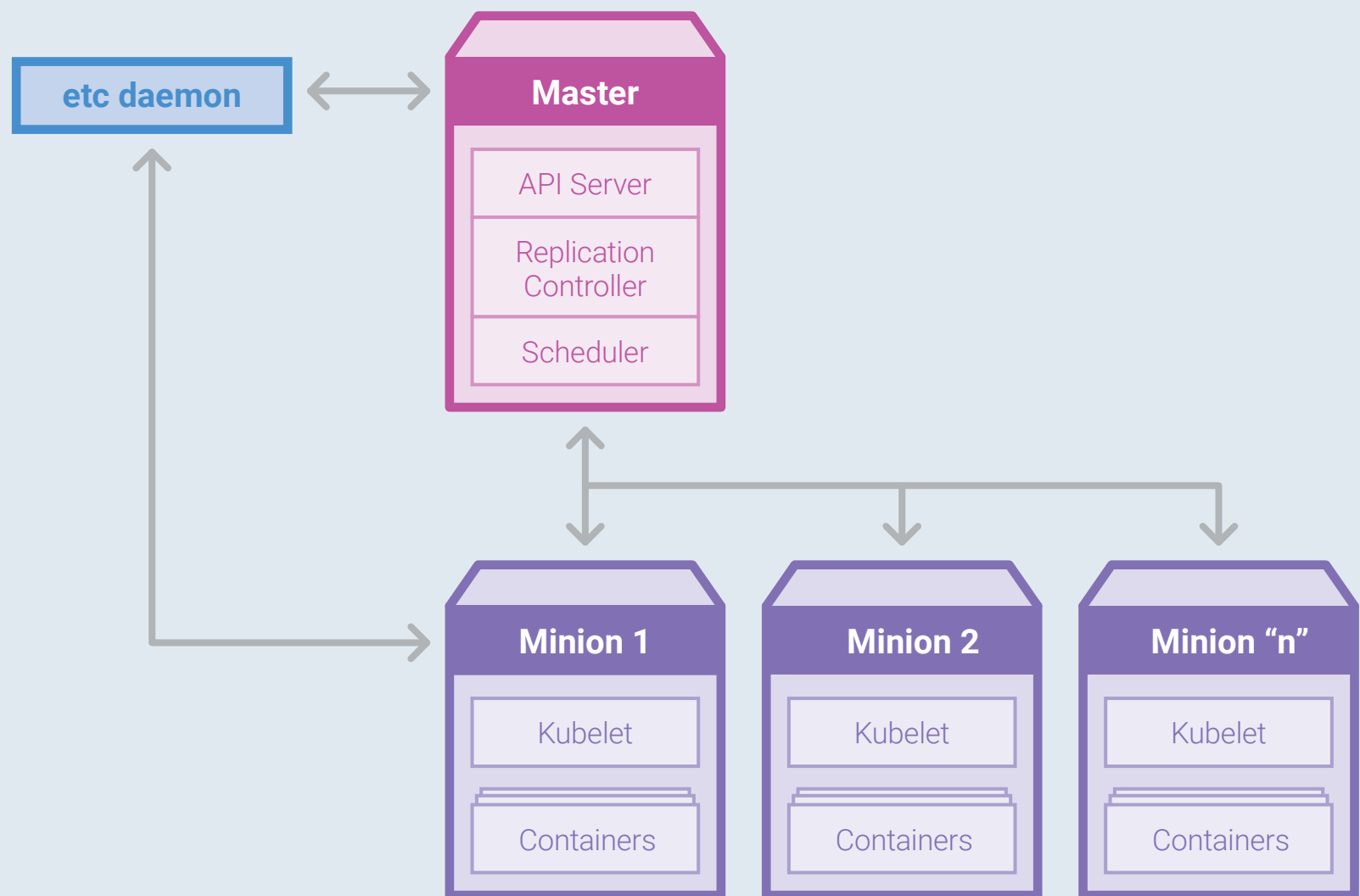
NOTE: Apache Mesos is a perfect orchestration tool for mixed environments with both containerized and non-containerized workloads. Although Apache Mesos is stable, many say it presents a steeper learning curve for container users.

Kubernetes

Originally an open source project launched by Google and now part of the Cloud Native Computing Foundation (CNCF), Kubernetes makes managing containers at web scale seamless with low operational overhead.

Kubernetes is not opinionated about the form or format of the container, and uses its own API and command-line interface (CLI) for container

Kubernetes: Building on Architectural Roots



Source: The New Stack

THE NEW STACK

FIG 1.3: *Kubernetes' relationship between the master and its nodes, still known in some circles as "minions."*

orchestration. It supports multiple container formats, including not just Docker's but also rkt, originally created by CoreOS, now a CNCF-hosted project. The system is also highly modular and easily customizable, allowing users to pick any scheduler, networking system, storage system, and set of monitoring tools. It starts with a single cluster, and may be extended to web scale seamlessly.

The six key features of an orchestrator that we mentioned earlier apply to Kubernetes in the following ways:

- **Scheduling:** The Kubernetes scheduler ensures that demands for resources placed upon the infrastructure may be met at all times.
- **Resource management:** In the context of Kubernetes, a resource is a logical construct that the orchestrator can instantiate and manage,

such as a service or an application deployment.

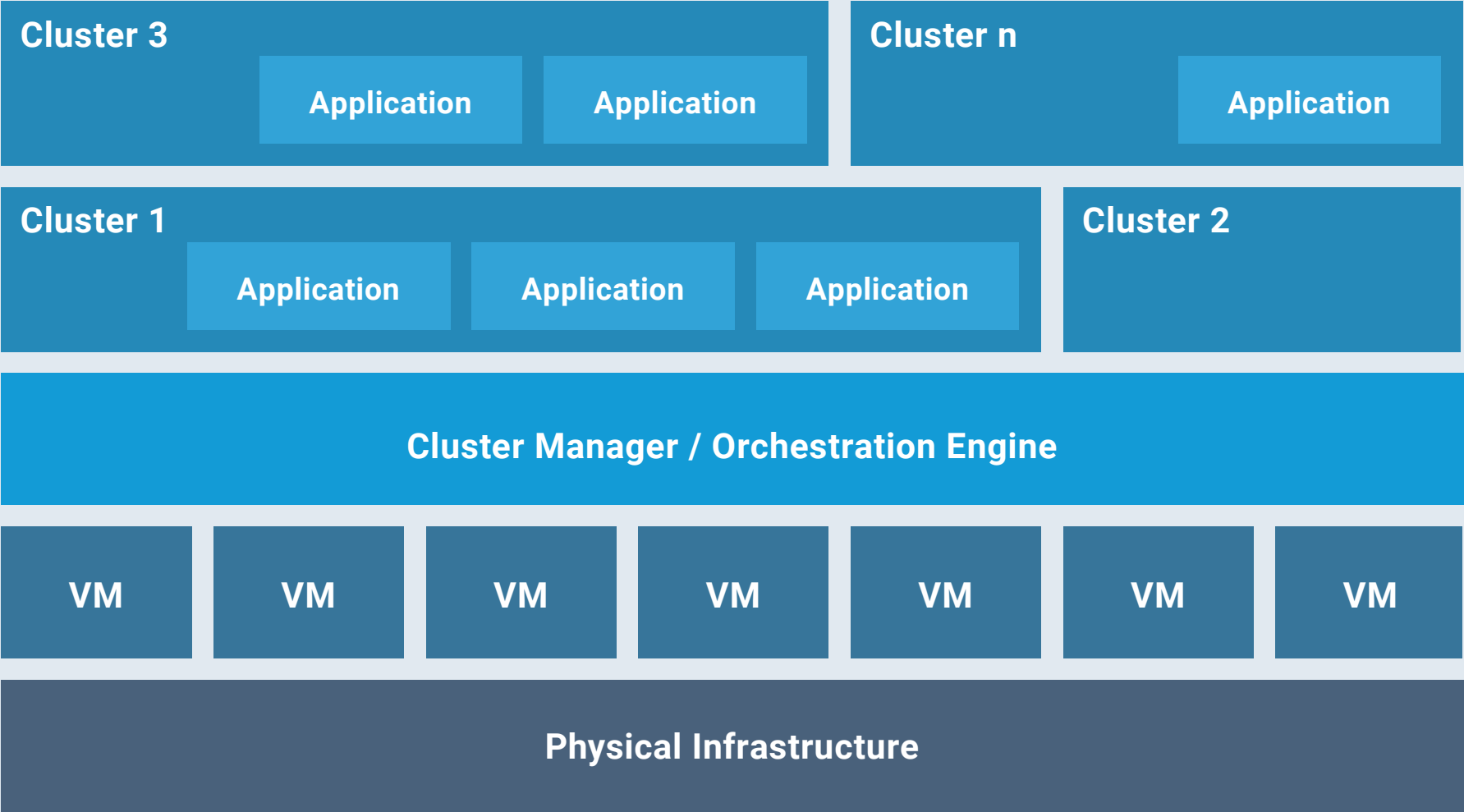
- **Service discovery:** Kubernetes enables services sharing the system together to be discoverable by name. This way, the pods containing services may be distributed throughout the physical infrastructure without having to retain network services to locate them.
- **Health check:** Kubernetes utilizes functions called “liveness probes” and “readiness probes” to provide periodic indications to the orchestrator of the status of applications.
- **Autoscaling:** With Kubernetes, the horizontal pod autoscaler automatically generates more replicas when it appears the designated CPU resources for a pod may be underutilized.
- **Updates/upgrades:** An automated, rolling upgrade system enables each Kubernetes deployment to remain current and stable.

NOTE: Kubernetes is built for web scale by a very vibrant community. It provides its users with more choices for extending the orchestration engine to suit their needs. Since it uses its own API, users more familiar with Docker will encounter somewhat of a learning curve.

Kubernetes Architecture

A contemporary application, packaged as a set of containers, needs an infrastructure robust enough to deal with the demands of clustering and the stress of dynamic orchestration. Such an infrastructure should provide primitives for scheduling, monitoring, upgrading and relocating containers across hosts. It must treat the underlying compute, storage, and network primitives as a pool of resources. Each containerized workload should be capable of taking advantage of the resources exposed to it, including CPU cores, storage units and networks.

Container Orchestration Engine



Source: Janakiram MSV

THE NEW STACK

FIG 1.4: *The resource layers of a system, from the perspective of the container orchestration engine.*

Kubernetes is an open source cluster manager that abstracts the underlying physical infrastructure, making it easier to run containerized applications at scale. An application, managed through the entirety of its life cycle by Kubernetes, is composed of containers gathered together as a set and coordinated into a single unit. An efficient cluster manager layer lets Kubernetes effectively decouple this application from its supporting infrastructure, as depicted in Figure 1.4. Once the Kubernetes infrastructure is fully configured, DevOps teams can focus on managing the deployed workloads instead of dealing with the underlying resource pool.

The Kubernetes API may be used to create the components that serve as the key building blocks, or primitives, of microservices. These components are autonomous, meaning that they exist independently from other components. They are designed to be loosely coupled, extensible and

adaptable to a wide variety of workloads. The API provides this extensibility to internal components, as well as extensions and containers running on Kubernetes.

Pod

The pod serves as Kubernetes' core unit of workload management, acting as the logical boundary for containers sharing the same context and resources. Grouping related containers into pods makes up for the configurational challenges introduced when containerization replaced first-generation virtualization, by making it possible to run multiple dependent processes together.

Each pod is a collection of one or more containers that use remote procedure calls (RPC) for communication, and that share the storage and networking stack. In scenarios where containers need to be coupled and co-located — for instance, a web server container and a cache container — they may easily be packaged in a single pod. A pod may be scaled out either manually, or through a policy defined by way of a feature called Horizontal Pod Autoscaling (HPA). Through this method, the number of containers packaged within the pod is increased proportionally.

Pods enable a functional separation between development and deployment. While developers focus on their code, operators can concentrate on the broader picture of which related containers may be stitched together into a functional unit. The result is the optimal amount of portability, since a pod is just a manifest of multiple container images managed together.

Service

The services model in Kubernetes relies upon the most basic, though most important, aspect of microservices: discovery.

A single pod or a replica set (explained in a moment) may be exposed to

internal or external clients via services, which associate a set of pods with a specific criterion. Any pod whose labels match the selector will automatically be discovered by the service. This architecture provides a flexible, loosely-coupled mechanism for service discovery.

When a pod is created, it is assigned an IP address accessible only within the cluster. But there is no guarantee that the pod's IP address will remain the same throughout its life cycle. Kubernetes may relocate or re-instantiate pods at runtime, resulting in a new IP address for the pod.

To compensate for this uncertainty, services ensure that traffic is always routed to the appropriate pod within the cluster, regardless of the node on which it is scheduled. Each service exposes an IP address, and may also expose a DNS endpoint, both of which will never change. Internal or external consumers that need to communicate with a set of pods will use the service's IP address, or its more generally known DNS endpoint. In this way, the service acts as the glue for connecting pods with other pods.

Service Discovery

Any API object in Kubernetes, including a node or a pod, may have key-value pairs associated with it — additional metadata for identifying and grouping objects sharing a common attribute or property. Kubernetes refers to these key-value pairs as labels.

A selector is a kind of criterion used to query Kubernetes objects that match a label value. This powerful technique enables loose coupling of objects. New objects may be generated whose labels match the selectors' value. Labels and selectors form the primary grouping mechanism in Kubernetes for identifying components to which an operation applies.

A replica set relies upon labels and selectors for determining which pods will participate in a scaling operation. At runtime, pods may be scaled by means of replica sets, ensuring that every deployment always runs the

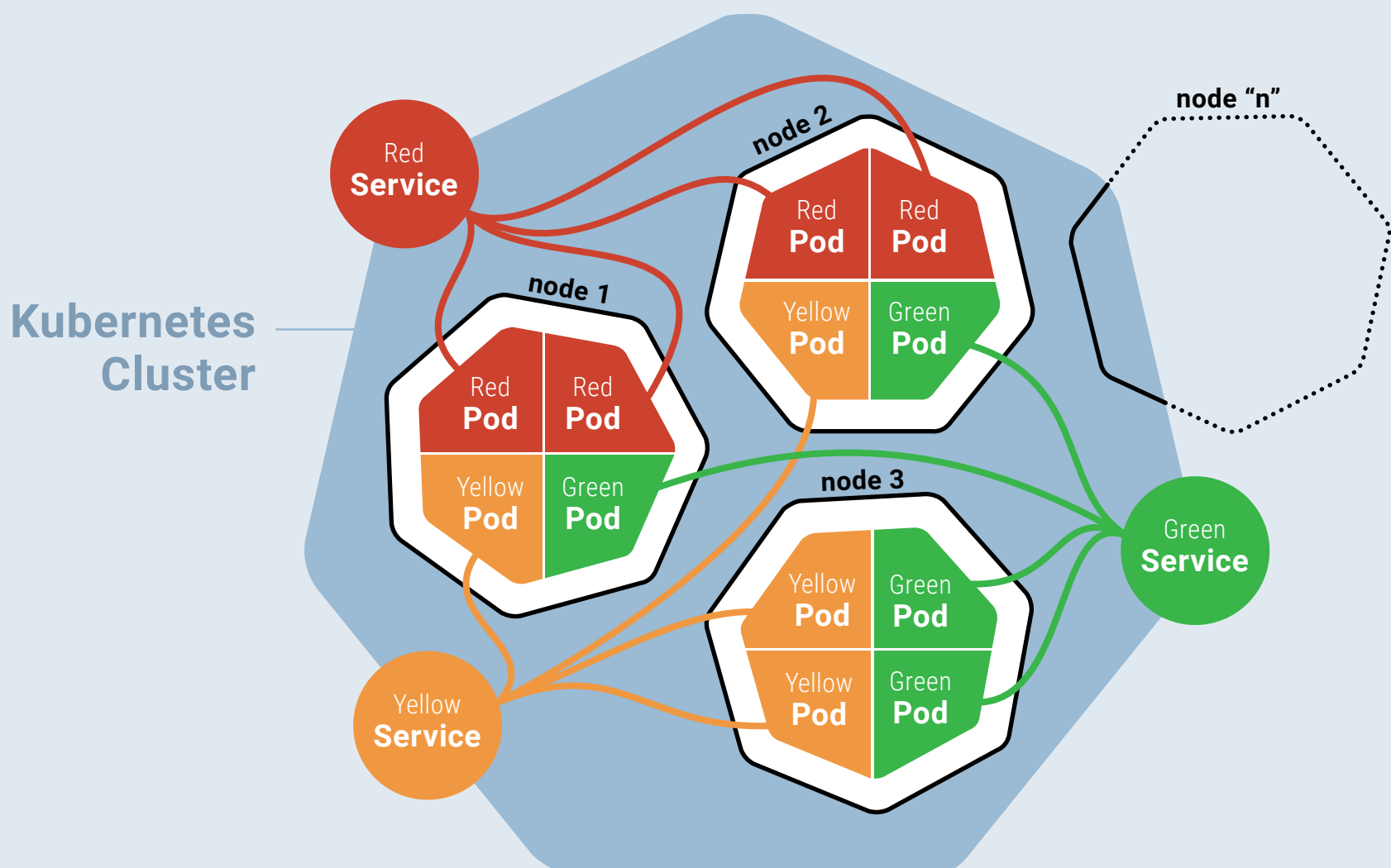
desired number of pods. Each replica set maintains a pre-defined set of pods at all times.

Any pod whose label matches the selector defined by the service will be exposed at its endpoint. When a scaling operation is initiated by a replica set, new pods created by that operation will instantly begin receiving traffic. A service then provides basic load balancing by routing traffic across matching pods.

Figure 1.5 depicts how service discovery works within a Kubernetes cluster. Here, there are three types of pods, represented by red, green and yellow boxes. A replication controller has scaled these pods to run instances across all the available nodes. Each class of pod is exposed to clients through a service, represented by colored circles. Assuming that each pod has a label in the form of `color=value`, its associated service

FIG 1.5: While a Kubernetes cluster focuses on pods, they're represented to the outside world by services.

How Services in a Cluster Map to Functions in Pods



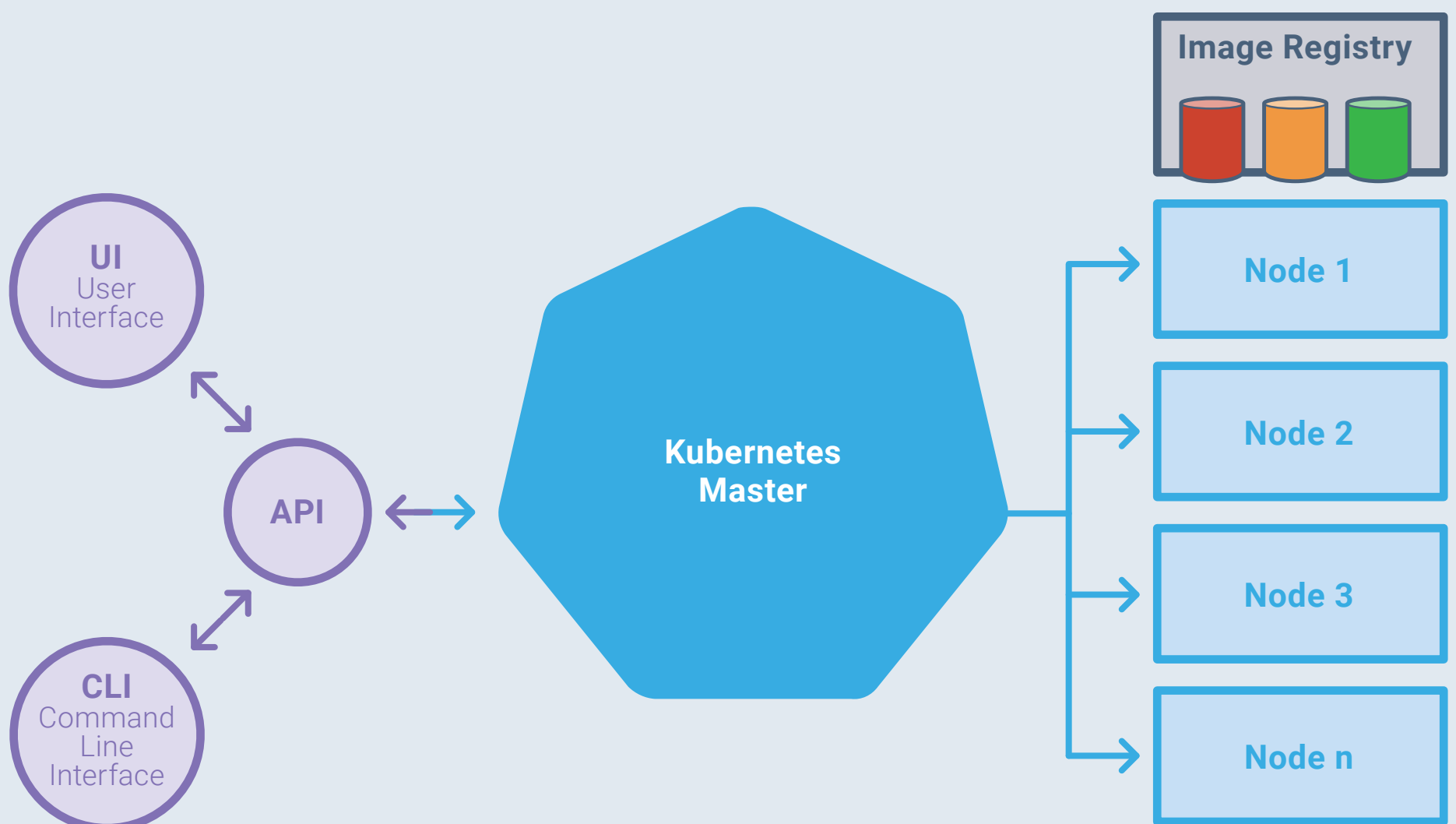
would have a selector that matches it.

When a client hits the red service, the request is routed to any of the pods that match the label `color=red`. If a new red pod is scheduled as a part of the scaling operation, it is immediately discovered by the service, by virtue of its matching label and selector.

Services may be configured to expose pods to internal and external consumers. An internally exposed service is available through a `ClusterIP` address, which is routable only within the cluster. Database pods and other sensitive resources that need not have external exposure are configured for `ClusterIP`. When a service needs to become accessible to the outside world, it may be exposed through a specific port on every node, which is called a `NodePort`. In public cloud environments, Kubernetes can provision a load balancer automatically configured for routing traffic to its nodes.

FIG 1.6: *The master's place in Kubernetes architecture.*

Kubernetes Architecture

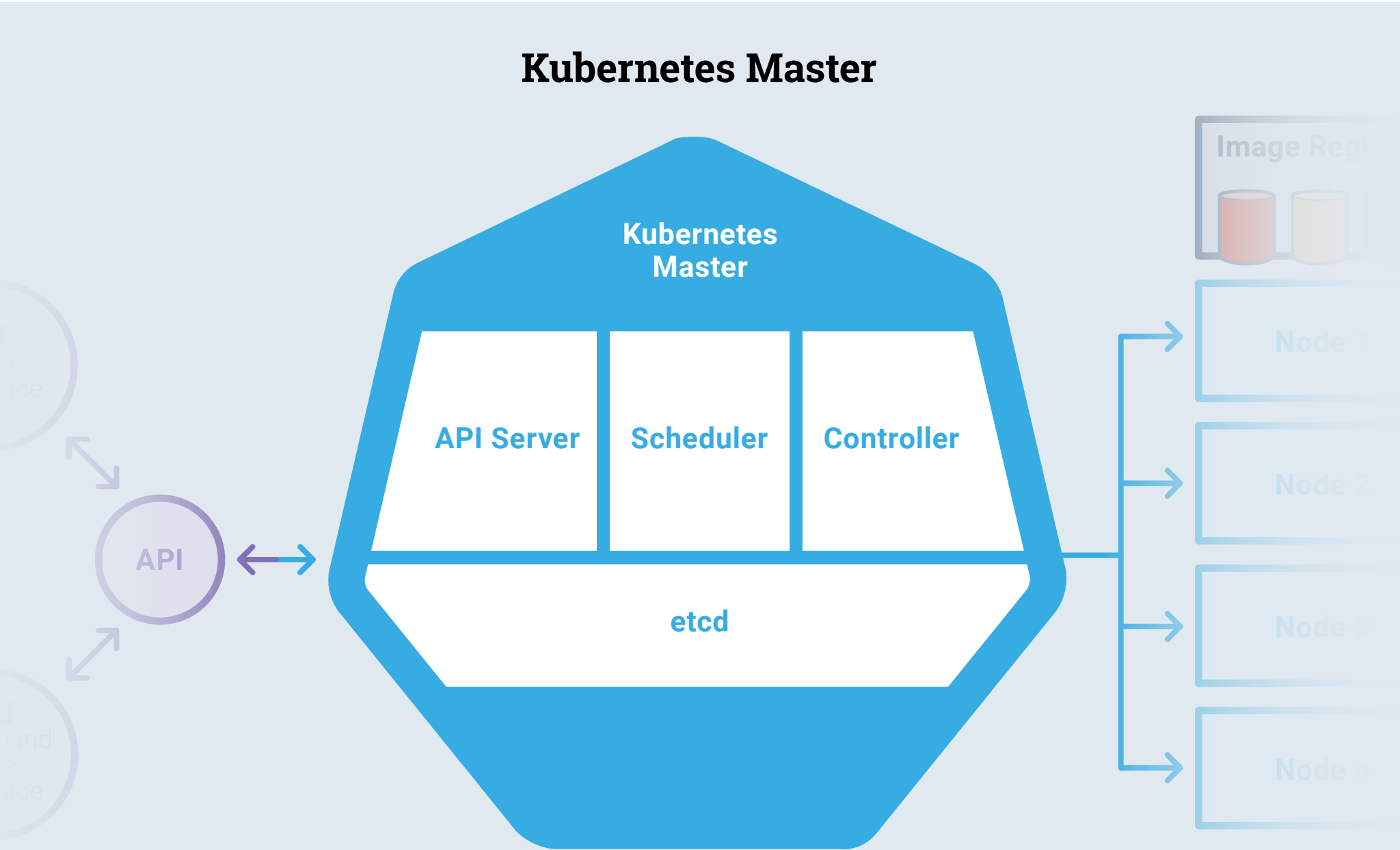


Master

Like most modern distributed computing platforms, Kubernetes utilizes a master/worker architecture. As Figure 1.6 shows, the master abstracts the nodes that run applications from the API with which the orchestrator communicates.

The master is responsible for exposing the Kubernetes API, scheduling the deployments of workloads, managing the cluster, and directing communications across the entire system. As depicted in Figure 1.6, the master monitors the containers running in each node as well as the health of all the registered nodes. Container images, which act as the deployable artifacts, must be available to the Kubernetes cluster through a private or public image registry. The nodes that are responsible for scheduling and running the applications access the images from the registry.

FIG 1.7: *The master’s place in Kubernetes architecture.*



As Figure 1.7 shows, the Kubernetes master runs the following components that form the control plane:

etcd

Developed by CoreOS, etcd is a persistent, lightweight, distributed, key-value data store that maintains the cluster's configuration data. It represents the overall state of the cluster at any given point of time, acting as the single source of truth. Various other components and services watch for changes to the etcd store to maintain the desired state of an application. That state is defined by a declarative policy — in effect, a document that states the optimum environment for that application, so the orchestrator can work to attain that environment. This policy defines how the orchestrator addresses the various properties of an application, such as the number of instances, storage requirements and resource allocation.

API Server

The API server exposes the Kubernetes API by means of JSON over HTTP, providing the REST interface for the orchestrator's internal and external endpoints. The CLI, the web UI, or another tool may issue a request to the API server. The server processes and validates the request, and then updates state of the API objects in etcd. This enables clients to configure workloads and containers across worker nodes.

Scheduler

The scheduler selects the node on which each pod should run based on its assessment of resource availability, and then tracks resource utilization to ensure the pod isn't exceeding its allocation. It maintains and tracks resource requirements, resource availability, and a variety of other user-provided constraints and policy directives; for example, quality of service (QoS), affinity/anti-affinity requirements and data locality. An operations team may define the resource model declaratively. The scheduler

interprets these declarations as instructions for provisioning and allocating the right set of resources to each workload.

Controller

The part of Kubernetes' architecture which gives it its versatility is the controller, which is a part of the master. The controller's responsibility is to ensure that the cluster maintains the desired state of configuration of nodes and pods all the time. By desired state we're referring to the balance of the utilized resources declared and requested by the pods' YAML configuration files, against the current demands and constraints of the system.

The controller maintains the stable state of nodes and pods by constantly monitoring the health of the cluster, and the workloads deployed on that cluster. For example, when a node becomes unhealthy, the pods running on that node may become inaccessible. In such a case, it's the job of the controller to schedule the same number of new pods in a different node. This activity ensures that the cluster is maintaining the expected state at any given point of time.

The Kubernetes controller plays a crucial role in running containerized workloads in production, making it possible for an organization to deploy and run containerized applications that go well beyond the typical stateless and scale-out scenarios. The controller manager oversees the core Kubernetes controllers:

- The [ReplicationController](#) ([ReplicaSet](#)) maintains the pod count of a specific deployment within the cluster. It guarantees that a given number of pods are running all the time in any of the nodes.
- [StatefulSet](#) is similar to replica set, but for pods that need persistence and a well-defined identifier.

- [DaemonSet](#) ensures that one or more containers packaged as a pod are running in each node of the cluster. This is a special type of controller that forces a pod to run on every node. The number of pods running as a part of **DaemonSet** is directly proportional to the number of nodes.
- The [job](#) and [cron job](#) controllers handle background processing and batch processing.

These controllers communicate with the API server to create, update and delete the resources that they manage, such as pods and service endpoints.

Mission critical applications require higher levels of resource availability. Through the use of a replica set, Kubernetes ensures that a predefined number of pods are running all the time. But these pods are stateless and ephemeral. It's very difficult to run stateful workloads, such as a database cluster or a big data stack, for the following reasons:

- Each pod is assigned an arbitrary name at runtime.
- A pod may be scheduled on any available node unless an affinity rule is in effect, in which case the pod may only be scheduled on nodes that have a specific label or labels specified in the rule.
- A pod may be restarted and relocated at any point of time.
- A pod may never be referenced directly by its name or IP address.

Starting with version 1.5, Kubernetes introduced the concept of stateful sets (represented by the **StatefulSets** object) for running highly available workloads. A stateful pod participating in a stateful set will have the following attributes:

- A stable host name that will always be resolved by the DNS.

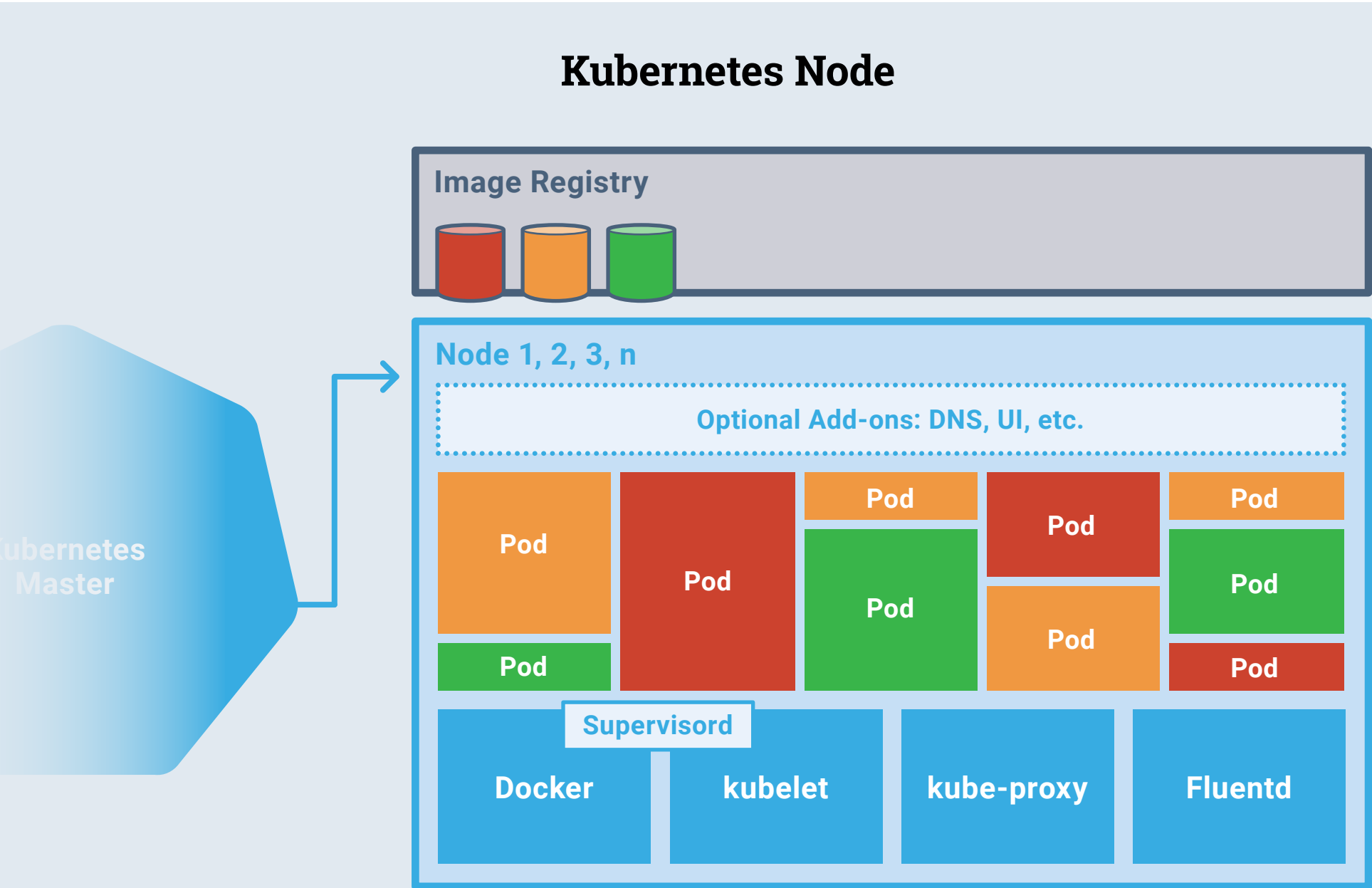
- An ordinal index number to represent the sequential placement of the pod in the replica set.
- A stable storage that is linked to the host’s name and ordinal index number.

The stable host name with its ordinal index number enables one pod to communicate with another in a predictable and consistent manner. This is the fundamental difference between a stateless pod and stateful pod.

Node

The node is the workhorse of the Kubernetes cluster, responsible for running containerized workloads; additional components of logging, monitoring and service discovery; and optional add-ons. Its purpose is to expose compute, networking and storage resources to applications. Each node includes a container runtime, such as Docker or rkt, plus an agent

FIG 1.8: An exploded view shows the multitude of components in a Kubernetes node.



that communicates with the master. A node may be a virtual machine (VM) running in a cloud or a bare metal server inside the data center.

Each node, as shown in Figure 1.8, contains the following:

Container Runtime

The container runtime is responsible for managing the life cycle of each container running in the node. After a pod is scheduled on the node, the runtime pulls the images specified by the pod from the registry. When a pod is terminated, the runtime kills the containers that belong to the pod. Kubernetes may communicate with any OCI-compliant container runtime, including Docker and rkt.

Kubelet

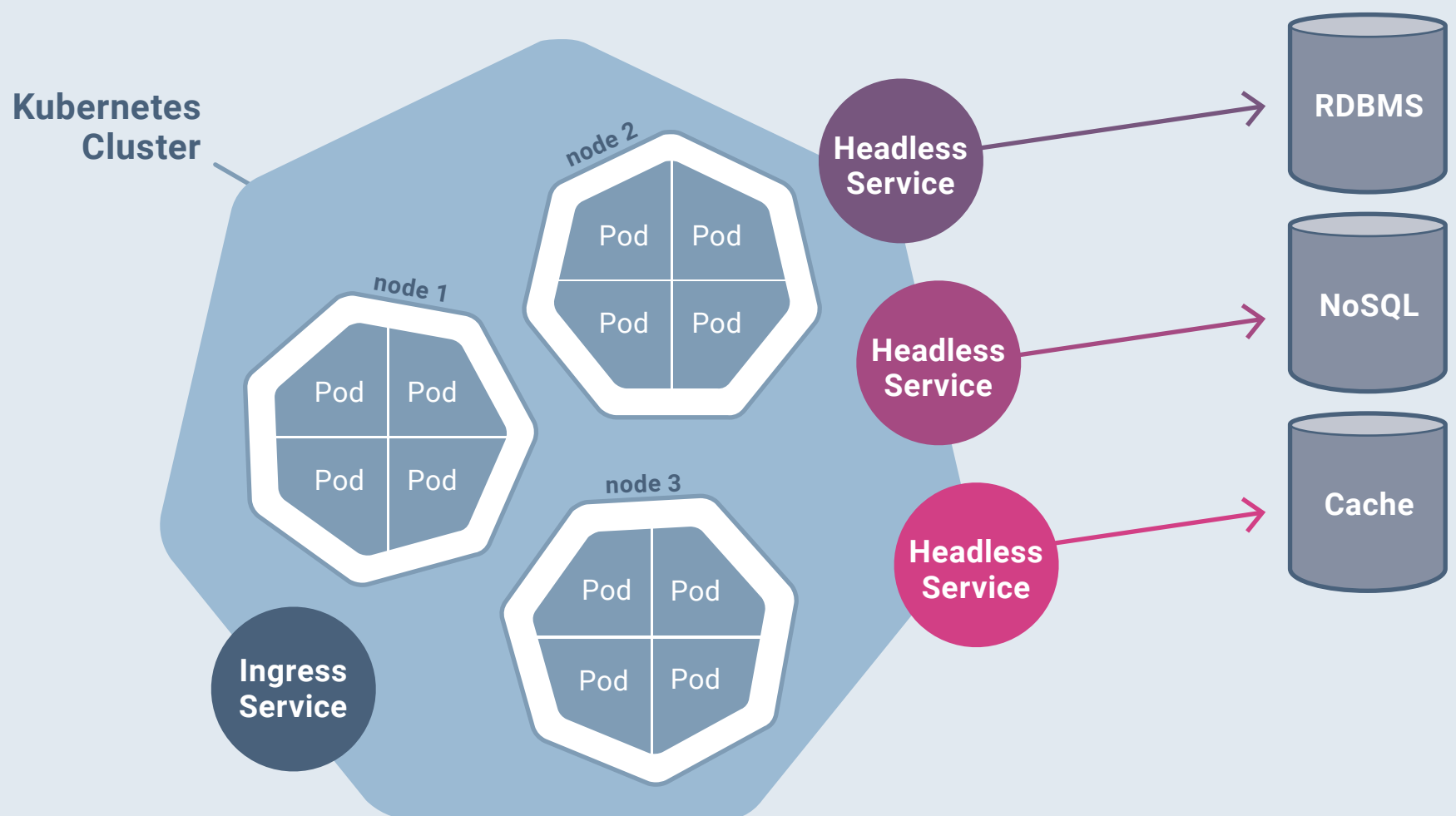
A kubelet is a component that ensures all containers on a node are healthy. It interfaces with the container runtime to perform operations such as starting, stopping and maintaining containers.

Each kubelet also monitors the state of pods. When a pod does not meet the desired state as defined by the replication controller, it may be restarted on the same node. The node's status is transmitted to the master every few seconds via heartbeat messages. If the master detects a node failure, the replication controller observes this state change and schedules the pods on other healthy nodes.

Kube-proxy

The **kube-proxy** component is implemented as a network proxy and a load balancer. It routes traffic to the appropriate container based on its IP address and the port number of an incoming request. It also takes advantage of OS-specific networking capabilities by manipulating the policies and rules defined through **iptables**. Each **kube-proxy** may be integrated with container-specific networking layers such as Flannel and Calico.

How Headless Services Attach to Functions



Source: <https://kubernetes.io/docs/concepts/services-networking/service/>

THE NEW STACK

FIG 1.9: Headless services provide connection points between stateful databases and services.

You can use something called a headless service to direct pods to external services such as cache, object storage and databases. As depicted in Figure 1.9, this is essentially the same thing as a service, but without the need for a **kube-proxy** or load balancing. A call to a headless service resolves to the IP address of the cluster that the service specifically selects. This way, you can employ custom logic for selecting the IP address, bypassing the normal route.

Logging Layer

The orchestrator makes frequent use of logging as a means for gathering resource usage and performance metrics for containers on each node, such as CPU, memory, file and network usage. The Cloud Native Computing Foundation produces what it calls a unified logging layer for use with Kubernetes or other orchestrators, called Fluentd. This

component produces metrics that the Kubernetes controller master needs to keep track of available cluster resources, as well as the health of the overall infrastructure.

Add-Ins

Kubernetes supports additional services in the form of add-ins. These optional services, such as DNS and dashboard, are deployed like other applications but integrated with other core components on the node such as Fluentd and `kube-proxy`. For example, the dashboard add-in pulls the metrics from Fluentd to display rich visualizations of resource utilization. DNS add-in augments `kube-proxy` through name resolution.

Distinguishing Kubernetes Platforms from One Another

Ever since cloud computing became the norm in modern enterprises, systems architects have been confused about the right balance of abstraction that a cloud platform should employ. The right balance would enable operational efficiencies, while at the same time improving developer productivity.

In the early days of cloud computing, the virtual machine was the basic unit of compute. End users were forced to select between so-called [IaaS+](#) platforms like AWS, and PaaS platforms like Heroku, Engine Yard, Cloud Foundry and OpenShift. The level of abstraction determined how much direct control a developer would have over the platform's underlying infrastructure. Greater abstraction would mean the developer would need to use the platform's API to push code.

Now that containers are emerging as the basic unit of compute, users are confronting similar questions about the right abstraction level for their needs. As the table below shows, there are many different types of

Kubernetes distributions in the container orchestration realm. The needs of the user — including the working environment, the availability of expertise, and the specific use case the user is dealing with — determine whether Containers as a Service (CaaS) or an abstracted platform is the right choice. No single, straightforward framework exists to guarantee a perfect decision. But Table 1.1 may be a start.

Types of Distributions				
Description	Community Supported	Vendor Distro (no value add)	Vendor Distro (with value add)	App Platforms / PaaS
Developer Abstraction	No	No	No	Yes
Target User	Cluster Operator	Cluster Operator / IT	Cluster Operator / IT	Cluster Operator / IT / Developers
Usage Maturity	Kicking tires / PoC	Production	Production	Production
Developer Tools / Middleware	No	No	Maybe	Yes
DevOps Tools	No	No	Maybe	Yes
Vanilla Distribution	Yes	Maybe	No	No
Control Over Environment	Yes	Yes	Yes	No
Benefits	Resource Optimization	Ops Productivity	Ops Productivity	Ops and Developer Productivity
Enterprise Support	No, Community Support only	Yes	Yes	Yes

TABLE 1.1: *A comparison between the types of Kubernetes distributions, ranging from fully community-produced to fully commercial.*

A CaaS platform includes the Kubernetes project, and also packages additional tooling needed for its deployment and management. An abstracted platform, on the other end of the scale, goes well beyond the operational efficiencies offered by CaaS, focusing on increased developer productivity. With CaaS, developers are required to package their own code in a container so that it may be deployed in a cluster. Even though Docker-based containers address this packaging problem on developers’

behalf, the abstracted application platforms can completely encapsulate the process of building container images internally — automating the process, as opposed to hand-holding the developer along the way. The developer's task stops once she has pushed her code to a source control tool like GitHub, or to a continuous integration / continuous delivery (CI/CD) system like Jenkins, and the platform does the rest.

By design, CaaS is closely aligned with the Kubernetes open source project, helping IT to run and manage containers at scale. But a CaaS model expects developers to take care of all of their applications' dependencies. From a cultural perspective, the DevOps model follows a culture of Dev and Ops working together with cross-functional knowledge. One of the things both teams have knowledge of, in this scenario, is the long list of dependencies.

Abstracted application platforms use Kubernetes as the core component, helping IT run containers at scale with less overhead than CaaS. Developers need not worry about managing runtimes or any application dependencies. Instead, they can focus on writing application code and pushing it to a source control repository or CI/CD system. Abstracted platforms enhance developer productivity, while taking away control over the underlying components. You can say DevOps is still at the center of this model. But with abstractions, there is no need for this cross-functional knowledge — it becomes redundant busy work. Developers need not understand the underpinnings of Kubernetes or how to manage it.

As we mentioned earlier, there is no straightforward framework for selecting between CaaS and abstracted platforms. Your choice depends on what your organization wants to achieve for developer productivity and the specific use cases you foresee for your organization.

Getting Started with Kubernetes

Here are some ways to get started with Kubernetes:

1. Sign up with a hosted CaaS service.
2. Download and install a single-processor setup for testing Kubernetes on a single PC, such as [CoreOS' Tectonic Sandbox](#).
3. Set up a local cluster by cloning the Kubernetes GitHub repo.
4. Try out a Kubernetes cluster for yourself right away with [Play with Kubernetes](#), which gives you a full Kubernetes environment you can run from your browser.

Public cloud providers, including IBM Bluemix, Google Cloud Platform and Microsoft Azure, offer hosted Kubernetes as a service. Through free credits and trial offers, it's easy to spin up a small Kubernetes cluster just for testing the waters. Refer to the documentation of the service of your choice for details on deploying your first application in Kubernetes.

Minikube is a single-node Kubernetes cluster that's ideal for learning and exploring the environment. It is strongly recommended for beginners with no past experience.

On a more powerful host machine, Kubernetes may be configured as a multi-node cluster based on Vagrant. The GitHub repo has instructions on setting up the Vagrant boxes.

PLOTTING THE KUBERNETES ROADMAP



In this podcast, Google assembled three of the members who oversee and contribute to the Roadmap project. In a containerized environment, there are still operating systems, and we still call most of the units that serve as vehicles for processes “Linux containers.” But inter-application communication (IAC) is no longer something that is facilitated by an underlying OS platform. As an orchestrator, Kubernetes facilitates the networking that takes place between components. How that facilitation will take place from here on out is a key topic of conversation for the people who assemble the Kubernetes Roadmap. [Listen to the Podcast.](#)



Aparna Sinha manages the product group at Google for Kubernetes. She started and co-leads the Kubernetes community PM SIG, which maintains the open source roadmap, and is a member of the CNCF Governing Board.



Eric Brewer leads Google’s compute infrastructure design, including Google Cloud Platform. As the long-time professor of computer science at the University of California, Berkeley, he has led projects on scalable servers, network infrastructure, IoT and the CAP Theorem.



Ihor Dvoretzkyi is an ambassador with CNCF, a product manager and OpenStack SIG head with the Kubernetes Community, and program manager at Mirantis. His focus has been on tight integration between Kubernetes and OpenStack.

KUBERNETES 1.7 AND EXTENSIBILITY



One of the people without whom Kubernetes would not exist, and perhaps the whole notion of orchestration would never have come to fruition, does not believe Kubernetes is truly at the center of the emerging ecosystem. Yes, we call it a “Kubernetes Ecosystem,” but Google’s Tim Hockin explained that he sees the platform as a hub for a greater kind of ecosystem to come.

“Part of what Kubernetes really set out to do at the beginning,” said Hockin, “was to provide the hub of ecosystems, plural. There’s the network ecosystem, the storage ecosystem and the security ecosystem.”

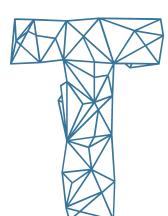
In this podcast, learn how the platform’s new extensibility model could enable integration with security policy, using whatever policy model you happen to have on hand. [Listen to the Podcast.](#)



Tim Hockin is a Principal Software Engineer at Google, where he works on Kubernetes and Google Container Engine (GKE). He is a co-founder of the Kubernetes project, and is responsible for topics including networking, storage, node, federation, resource isolation and cluster sharing.

MAP OF THE KUBERNETES ECOSYSTEM

by **SCOTT M. FULTON III**

 The term ecosystem was first applied to computing in the late 1970s, as an analogy to explain what was happening to the emerging field of Apple II software. Up to that point in time, nearly all software was exclusively distributed by the manufacturer of the computer for which it was written. But Apple chose to accept the presence of independent software vendors, some of which packaged their floppy diskettes in zipper bags with instructions printed on colored construction paper. When the computer manufacturer realized the relationships between all the parties, including itself, were mutually beneficial to everyone, Apple enabled the first ecosystem in our little hemisphere.

Nearly four decades later, it's the goal of most every computing platform to generate its own ecosystem — a kind of economy that coalesces around a product, where the people who create things that support that product, are in turn supported by it. The ethic of many of the leaders in the Kubernetes community is that the imposition of exclusivity is the antithesis of support.

No Point in Reinventing the Wheel

“Kubernetes is the platform. Kubernetes should never take an opinionated position about which monitoring solution you use,” said [Tim Hockin](#), Google’s principal software engineer. “We should be, and have been — and I’m committed to continuing to be — completely neutral when it comes to such decisions. I think it would be untenable to go to potential adoptees, and say, ‘Please not only install this Kubernetes thing (which is sort of a handful on its own) but also change all of your monitoring over to [Prometheus](#), and change all of your logging over to [Fluentd](#) — and oh, by the way, you also have to use [gRPC](#) and [OpenTracing](#).’ There’s no way that could work.”

“Kubernetes is a pluggable system, very much on purpose,” Hockin continued, “so that you can integrate multiple solutions into it. Now, being that things like Prometheus are part of the family, I would hope that people would look at Prometheus as sort of the cloud-native way of doing things. And if they don’t have an existing monitoring solution, they might look at the other things the CNCF [Cloud Native Computing Foundation] is doing, and consider those technologies. But I really mean ‘consider,’ and not, ‘be forced to adopt.’”

The CNCF does produce its own ecosystem map, called the [Cloud Native Landscape](#). Docker is easily locatable on this map, and one can find Kubernetes seated at the “Scheduling & Orchestration” table along with Docker Swarm, Mesos, Nomad, and Amazon Elastic Container Service (now EC2 Container Service).

But our purpose with this particular chapter is to chart the state of evolution, at the time of this publication, of the ecosystem that has thus far coalesced specifically around Kubernetes. One can make a potent argument that the container ecosystem (about which The New Stack has

already [published an ebook](#)) was catalyzed almost entirely by Docker. Though many organizations claim to have contributed to the concept of compartmentalized namespaces or portable workloads, only one is easily recognizable by its big, blue whale logo.

Kubernetes did not really invent workload scheduling and orchestration. But it did premiere a unique approach to the concept that enabled inter-process communication and scalability at a level, and with an ease of implementation, that Docker did not accomplish on its own.

For a plurality of enterprises to accept Kubernetes as a platform and, to a broader extent, as a methodology — as opposed to just another tool — it truly does need to be perceived as the progenitor of a broad set of interchangeable tools. Tim Hockin’s point is that the platform cannot afford to be perceived as attempting to institute a single way of work, through a single toolset. It may as well be an operating system, with certified software and licensed extension libraries, if it intended to do that.

[Laura Frank](#), the director of engineering at CI/CD platform provider Codeship, takes this idea a step further: She believes the focal point of an ecosystem cannot expect developers to fuel it first through their own homemade tools, before they can reap the benefits from it.

“My pragmatism is always to use a tool that exists, versus trying to write my own,” Frank told us. “To be very honest, with the rolling-update, self-healing nature of Kubernetes, I couldn’t write a better solution. I think there’s no point in trying to reinvent the wheel. I think it’s fine to rely on Kubernetes for things like that, instead of trying to automate them, or even write them yourself. I think it’s great to rely on tools — and especially open source tools, because I think the point about, ‘You’re not Google; you don’t have Google-sized problems,’ isn’t really that valid anymore.

“Kubernetes has such a rich ecosystem of open source community around

it, that it is not just representative of one company's interests," she continued. "It's really a collaboration across many different industries, and sizes of engineering teams and organizations coming together, to create something that works together the best way that it can, for the largest number of use cases that it can."

The DevOps Pipeline

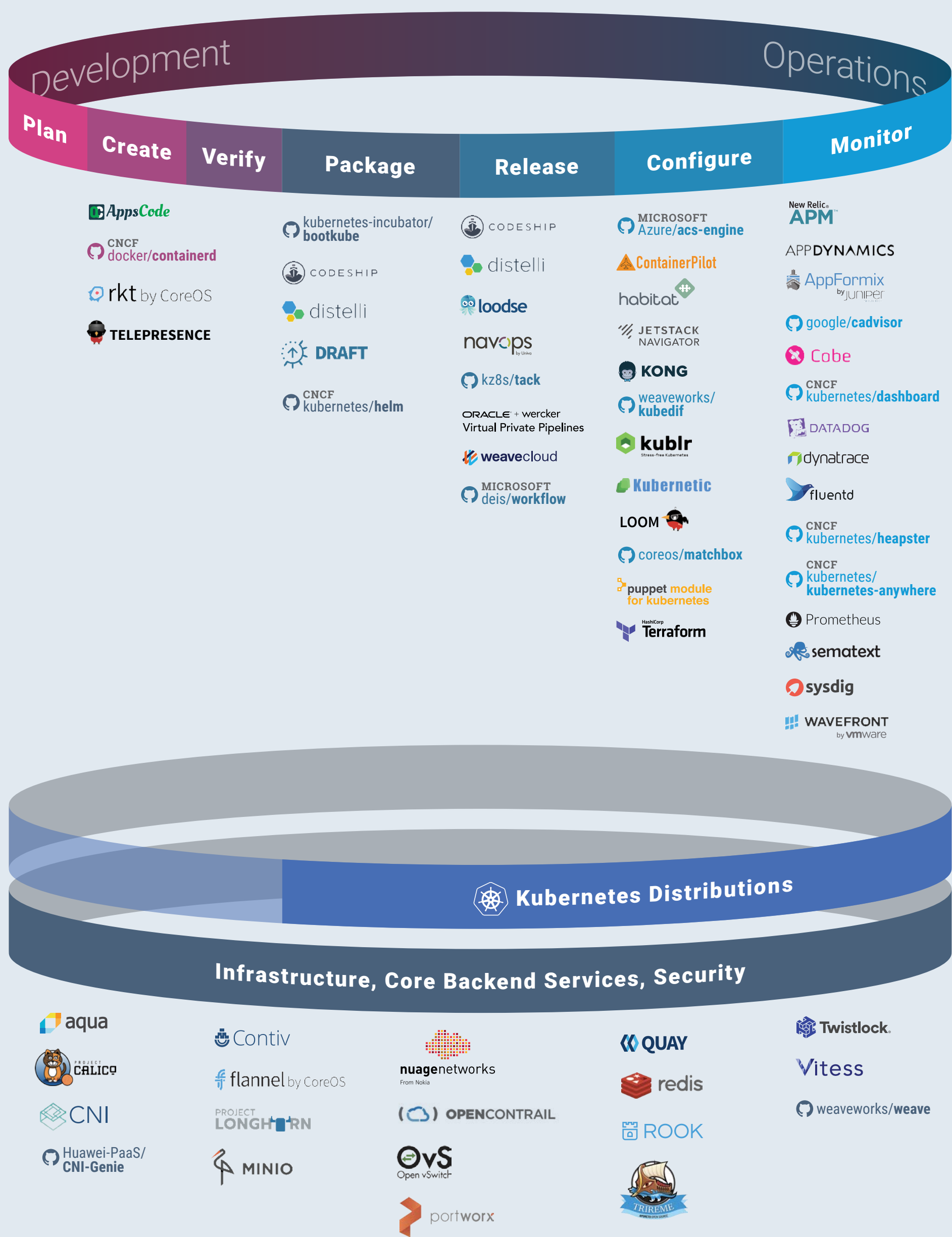
Figure 2.1 presents our depiction of the present state of the Kubernetes ecosystem.

It is a little lopsided. That's not an accident, and no, those omissions are not because we rushed this ebook to press. While many are quick to jump on the metaphor train and declare Kubernetes a complete DevOps life cycle management framework, when we partition the DevOps pipeline so that developer-centered tools gravitate to the left, and operations-centered tools to the right, the propensity of tools in the ecosystem lean to the right.

This is not some sort of ecosystem defect. One does not actually create and deploy a microservice with Kubernetes, the way one does not create an egg and deploy it with a skillet. There are plenty of independent development environments for that purpose. And there is Docker for packaging the product of that development in portable containers. The Kubernetes community is working on a toolset called [CRI-O](#) for staging pre-existing containers in a standard engine. But it does not take the place of Docker; CRI-O does not create container images.

The items we listed under the "Create" column of the pipeline include [Telepresence](#), a local development environment for building microservices for staging in Kubernetes; as well as the [rkt](#) and [containerd](#) runtimes, both of which are now CNCF projects. Codeship enters the picture at the

DevOps Pipeline for Kubernetes



Source: The New Stack

THE NEW STACK

FIG 2.1: Today's Kubernetes DevOps pipeline.

“Package” and “Release” stages of the pipeline, when you’re integrating microservices with continuous deployment patterns.

In the “Configure” column is where the management platforms come in: the systems that are not end-to-end life cycle management systems like OpenShift, but are nonetheless presented to customers as value-adds that go above and beyond what Kubernetes offers by itself. Then the “Monitor” column shows the many logging, oversight, and monitoring tools and platforms that may either be absorbed into the Kubernetes system (for instance, the Fluentd unified logging layer) or that plug into Kubernetes from the outside (e.g., [New Relic APM](#), [AppDynamics](#), [Dynatrace](#), [Sysdig](#), [VMware Wavefront](#)). Many of these monitoring platforms pre-date Kubernetes by several years, though they have certainly been brought into the platform’s sphere of influence.

At the base of the chart in Figure 2.1 are the infrastructure services that may support Kubernetes, such as the [Quay](#) container deployment platform, Nuage Networks’ [SDN](#), and the [Twistlock](#) container security platform.

“There’s really a few different, fundamental differences between securing containers and securing [virtual machines],” said Twistlock Chief Technology Officer [John Morello](#). “There’s a lot of misplaced writings out there that portray containers as the next evolution of virtual machines. And while they definitely share some similarities, the use cases and just the core technology is dramatically different. If you go into it thinking that the container is a sort of miniature virtual machine, you’re going to come into it with a set of biases and assumptions that really are not correct.”

Morello observes that a containerized, distributed application is an abundance of small entities — a plurality of objects. By contrast, from the developer’s perspective, the application under construction may still be a

single entity. And this may explain some of the lopsidedness in the appearance of today’s Kubernetes DevOps pipeline: By the time the application gets to the production stage, it becomes less of a construct and more of a population.

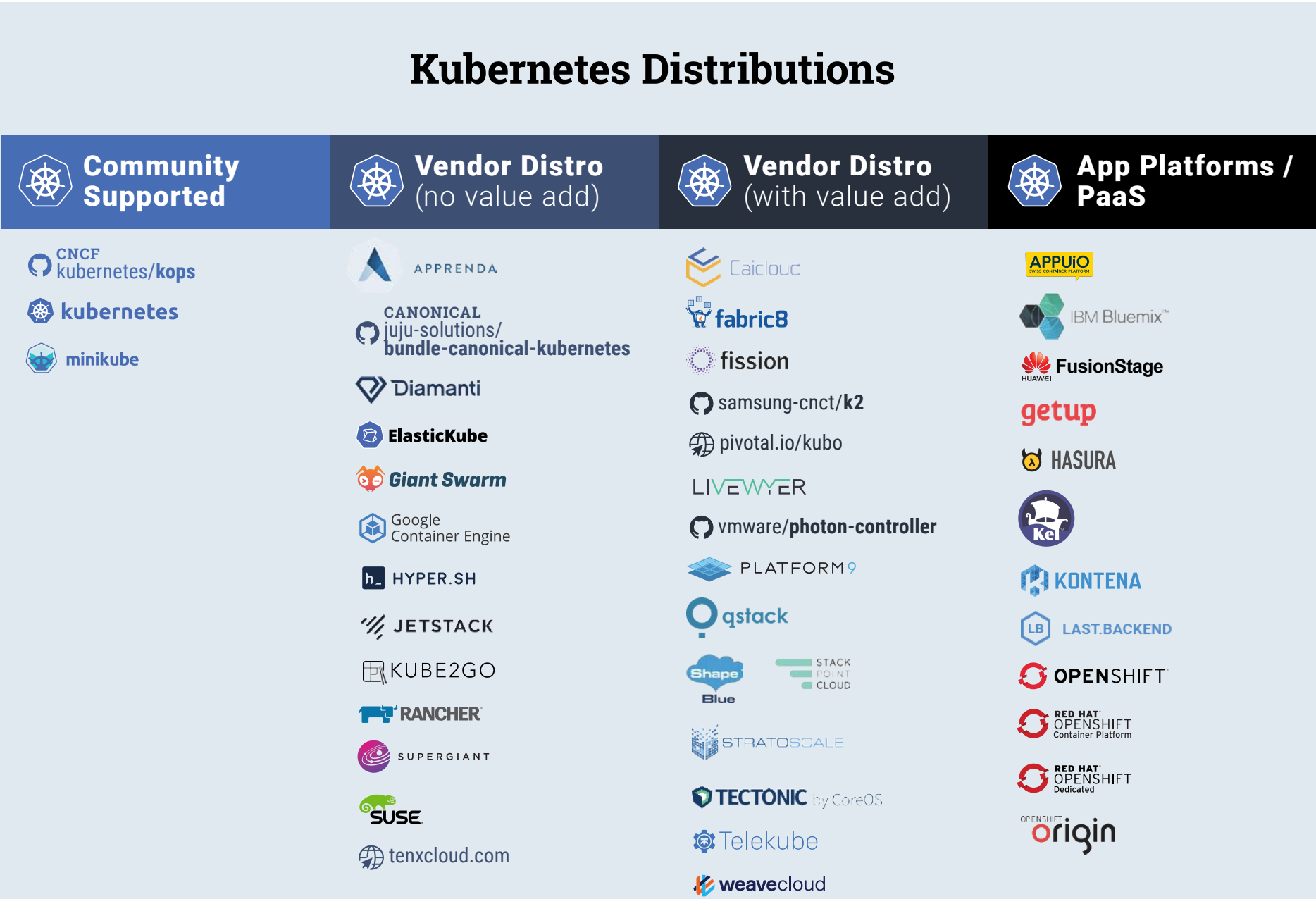
Levels of Support

Figure 2.2 lists the available packages and services through which you are able to obtain, or purchase, or purchase support for, Kubernetes.

We divide the classes as follows:

- **Community Supported** distributions are the free and open source packages with which you can deploy the platform yourself and give it a go. [Minikube](#) is a genuine Kubernetes environment for local, experimental deployments on a single computer.

FIG 2.2: The classes of Kubernetes platform distributions.



- **Vendor Distro (no value add)** shows companies that deliver software solutions, or cloud-based platforms, that include pure Kubernetes plus vendor-backed support.
- **Vendor Distro (with value add)** shows companies that offer more complete environments, including scheduling, development, and life cycle management, featuring Kubernetes at the core of their systems.
- **App Platforms / PaaS** shows full-scale Platforms as a Service, including [Red Hat OpenShift](#), that effectively abstract away the management and maintenance of Kubernetes behind end-to-end, automated development and deployment environments.

For those interested in an even more detailed collection, take a look at this frequently updated, community-managed [list of Kubernetes distributions](#).

Red Hat's product manager for Kubernetes and OpenShift, [Brian Gracely](#), told us he expects customers' recognition of Kubernetes to actually subside, as momentum for the platforms that contain it (such as his own, naturally) continue to rise.

"Ultimately, it kind of comes down to, are you building a business model around the idea that eventually, it does become boring?" asked Gracely. "You have a choice of then building some differentiation on top of that, or around that. What we see right now is a very, very healthy ecosystem around a lot of companies that want to be part of this digital transformation age — cloud-native applications — and that's fantastic. We're seeing people approach it from different perspectives. Red Hat is delivering it as software; we also deliver it as a service through OpenShift Online. We're seeing the major cloud providers taking it, and making it a service that anybody can consume — which is a new business model, and it's great.

“I think it’s perfectly fine to say the Kubernetes ecosystem right now is still in its early days,” he continued. “There’s still a lot of exploding innovation going on, and really cool ideas. But if you talk to the people who are dedicating a lot of engineering talent to this — Red Hat, Microsoft, Google, Intel, Huawei, IBM — ultimately, our goal is, we want the core technology to be stable. People are going to go to market in different ways. And yes, five years from now, maybe we won’t call it the ‘Kubernetes Ecosystem.’ But if I’m somebody who’s looking at Kubernetes, and people are saying, ‘Well, it’s becoming kind of a boring technology,’ that’s perfectly fine. It makes it much more tangible for a business customer to say, ‘This is something I don’t have to think about nearly as much.’”

ORCHESTRATION AND THE DEVELOPER CULTURE



In the young history of container orchestration, one of the most successful practitioners of merging the microservices mindset with enterprises' development and deployment pipelines has been the Berlin-based director of engineering for CI/CD platform provider Codeship, Laura Frank.

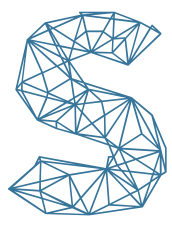
The more businesses Frank has come into contact with as a Docker Captain, the more she has come to understand the cultural issues facing developers in organizations today, and the extent to which platforms like Kubernetes may stop short of catalyzing revolutionary changes in and of themselves. In this podcast, hear more about “Captain” Frank’s experiences with businesses that are trying to navigate these new and still-uncharted waters. [Listen to the Podcast.](#)



Laura Frank is the director of engineering at Codeship and a Docker captain. Her primary focus has been on fortifying Docker’s infrastructure and improving its overall CI/CD experience. Prior to her involvement with Docker, she worked on HPE’s public cloud offering and on the OpenStack project.

USER EXPERIENCE SURVEY

by **LAWRENCE HECHT**



Some folks speculate that Kubernetes has already won the container orchestration wars. If indeed there are such wars, beyond what certain technology publications tend to apply to markets with any degree of contention whatsoever, then it's hard to deny that Kubernetes does seem to have an edge. Many early adopters have concluded that its combination of platform features and community gave it an edge over alternatives.

The New Stack's Kubernetes User Experience Survey stops short of conclusively proving Kubernetes has become, or is becoming, the de facto container orchestration tool among organizations we surveyed. Still, it provides a clear picture of how people are evaluating and deploying Kubernetes, the challenges they face, and their views on several competitive products. The survey also looked at container users to determine their selection criteria for orchestration platforms.

Key Findings

- **Production-ready:** Kubernetes adoption has increased as more containerized applications move into production.

- **The door remains open:** There may yet be a chance for Kubernetes' competitors. Whether Docker Swarm, AWS EC2 Container Service (ECS), Mesosphere Enterprise DC/OS, or any other product actually mounts a successful challenge to Kubernetes in this space, will depend largely upon how satisfied Kubernetes users are as a whole, and to a lesser extent upon other factors such as technological superiority and commercial value-adds.
- **Greater expectations:** You will find users who do complain about what they characterize as the platform's complexity, implementation difficulties, and maintenance headaches. Distributed systems are inherently complex, and Kubernetes is certainly no exception. But at least for now, users are generally happy with Kubernetes' approach.
- **Commercial platforms abound:** Some 45 percent of Kubernetes users surveyed have in place a vendor distribution, whether it be integrated into a platform, supplemented with additional software or just bundled with enterprise support. Whether they're tagged as Platform as Service (PaaS) or Container as a Service (CaaS), platforms like OpenShift (arguably a PaaS) and Google Container Engine (a CaaS platform) are becoming more popular. Vendors offering the standard fare of support with implementing or managing the platform may face less success unless they opt for something more unique and differentiated.
- **The simpler, the better:** [Flannel](#) is the leading network overlay for software-defined networking with Kubernetes, among those we surveyed. [Project Calico](#), the open source Border Gateway Protocol (BGP) router project, is also seeing significant adoption among cluster operators that have just begun implementing Kubernetes in production.

- **New monitoring methods take shape:** [Prometheus](#) is by far the most cited tool among our survey respondents for monitoring Kubernetes clusters. [Heapster](#), however, has also gained significant adoption among our group. Traditional monitoring vendors are not faring as well, although usage levels for their tools appear to increase when they are being integrated into a larger, custom monitoring platform.
- **Bringing containerization to operators:** Among the organizations we surveyed, the cluster operator job role is the one most likely to be implementing Kubernetes. Although application operators are less often involved, they may eventually have the most to gain when the platform is deployed as part of a continuous deployment strategy.

Sample and Methodology

We drew our observations and conclusions for this chapter from a web-based survey conducted by The New Stack from May 15 through May 29, 2017. Our report is based on the subset of data from 470 individuals who identified their organizations as container users. People with hands-on experience implementing Kubernetes were asked the most questions, although some questions were asked of other Kubernetes evaluators to ascertain their opinions and observations.

We made extra efforts to distinguish the responses of actual Kubernetes end users from those of members of the infrastructure and tools market. If a respondent's employer provided PaaS, Infrastructure as a Service (IaaS), or software deployment tools, we asked that person to limit responses to the use or evaluation of Kubernetes within the employer company. We'll tell you when vendor responses may have affected our findings.

Over half of our sample is composed of people employed or contracted by

enterprises with over 100 employees, with 24 percent working for companies with more than 1,000 employees. We'll also tell you when company size may have a material impact on our findings.

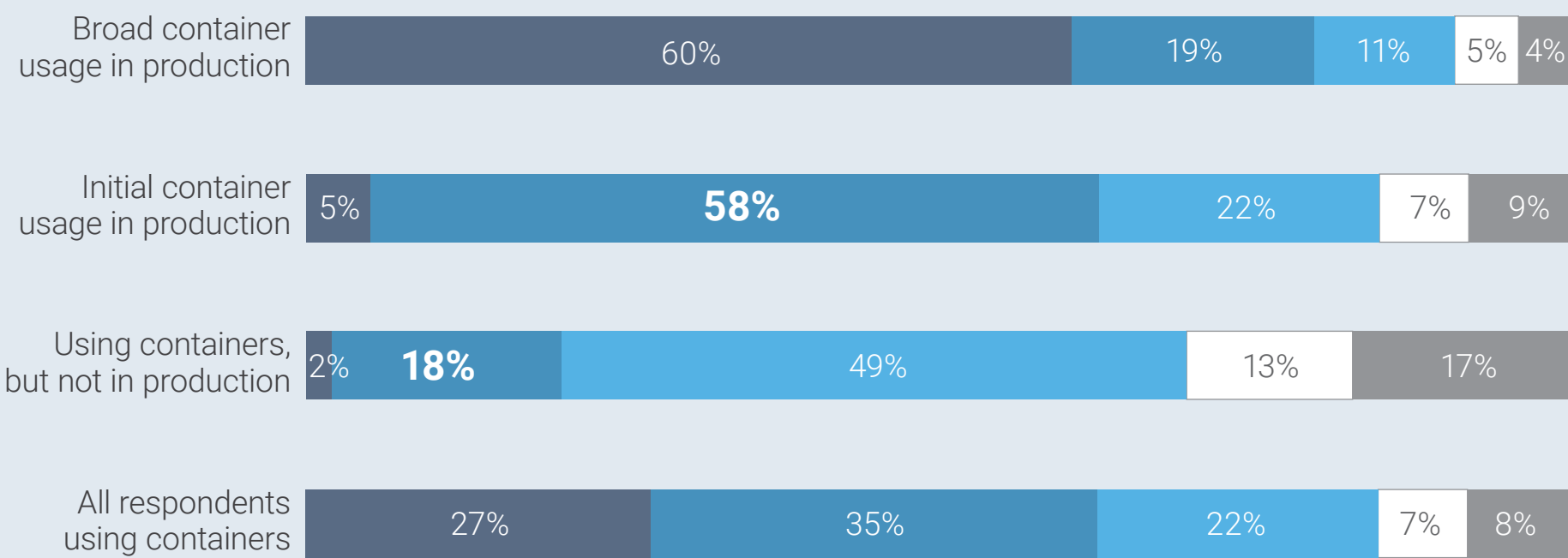
Respondents to our web survey are mainly users of virtual infrastructure and distributed systems platforms, which may or may not be representative of the broader IT market.

[A cleaned-up version of the data](#) is available for the community to review.

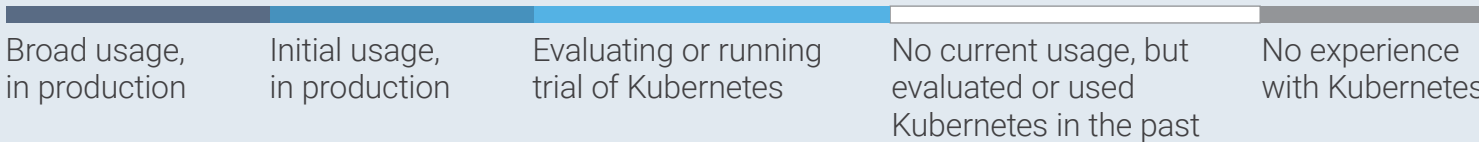
Who Uses Kubernetes

As we reported in The New Stack's 2016 survey (see "[The Present State of Container Orchestration](#)"), people don't realize their organizations need container orchestration until they've already deployed their first containerized applications to production. This event appears to be the motivating reason for respondents to identify and deploy tools to manage

Broad Kubernetes Adoption Follows Broad Container Adoption



Status of Kubernetes Usage



the infrastructure on which their containers are deployed.

We asked developers to characterize the current state of their organization's deployment of containers. Keep in mind, all of these developers' organizations are using containers to some extent. Among all developers surveyed, some 27 percent of respondents stated their organizations have already adopted Kubernetes broadly in production. Another 35 percent said their organizations are in the initial stages of Kubernetes adoption.

However, when we break down responses to this line of inquiry by the state of their production environments, a meager five percent of respondents whose organizations are in the initial stages of container use have adopted Kubernetes broadly. Nearly three in five, however, have begun their Kubernetes adoption. Compare this to only 18 percent of respondents whose organizations do use containers, though not yet in production, who are in the initial stages of using Kubernetes. This is a clear signal that the journey toward Kubernetes begins for most firms well after they've adopted containers thoroughly in production.

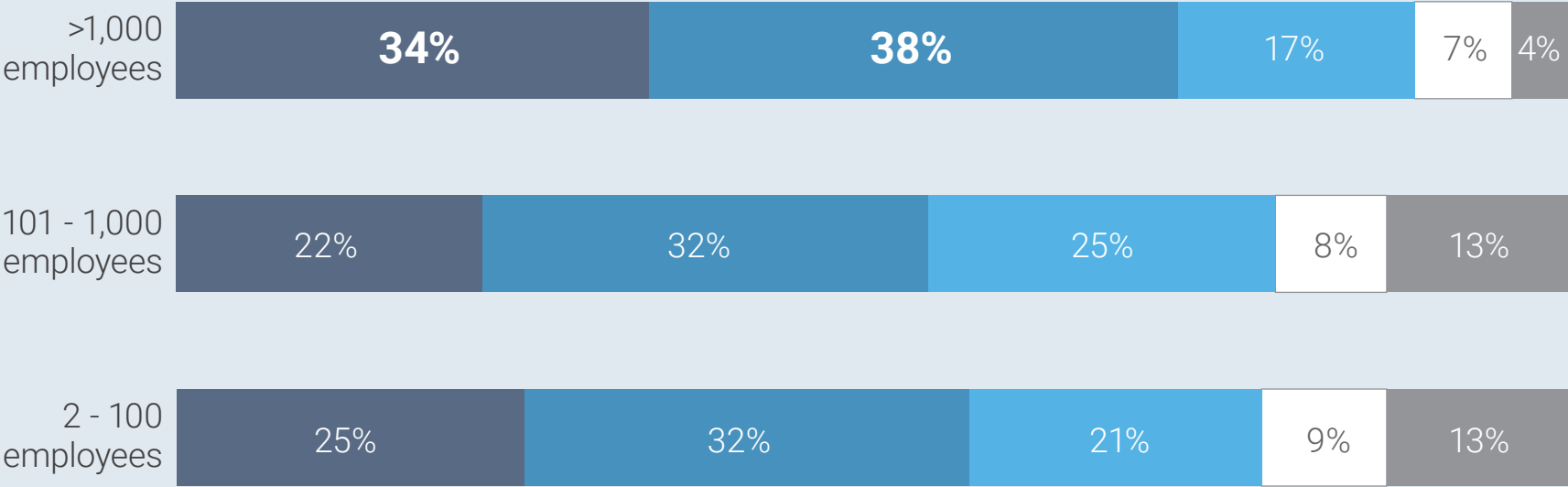
Put another way, if containers have not made their way to production use, don't expect Kubernetes to have been implemented.

Do Large Enterprises Prefer Kubernetes?

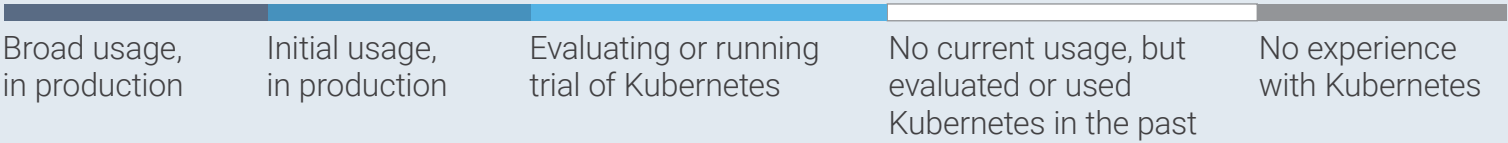
We gave respondents a list of five phrases, and asked them to choose the one they felt best characterized the relative state of the Kubernetes deployments in their own organizations.

Respondents working in large companies are more likely to be Kubernetes users. When asked to characterize the status of Kubernetes usage in their organizations, over one-third of respondents (34 percent) said they were in the deployment stage, with another 38 percent in the initial phase.

Kubernetes Implementation by Size of Enterprise/Organization



Status of Kubernetes Usage



Source: The New Stack 2017 Kubernetes User Experience Survey.
Q. What is the status of Kubernetes usage at your enterprise or organization?
Q. How many employees work at your enterprise or organization? n=333.
2-100 employees, n=118; 101-1,000 employees, n=72; > 1,000 employees, n=96.

THE NEW STACK

By comparison, only 22 percent of respondents in medium-sized organizations and some 25 percent of those in small companies had broadly implemented Kubernetes.

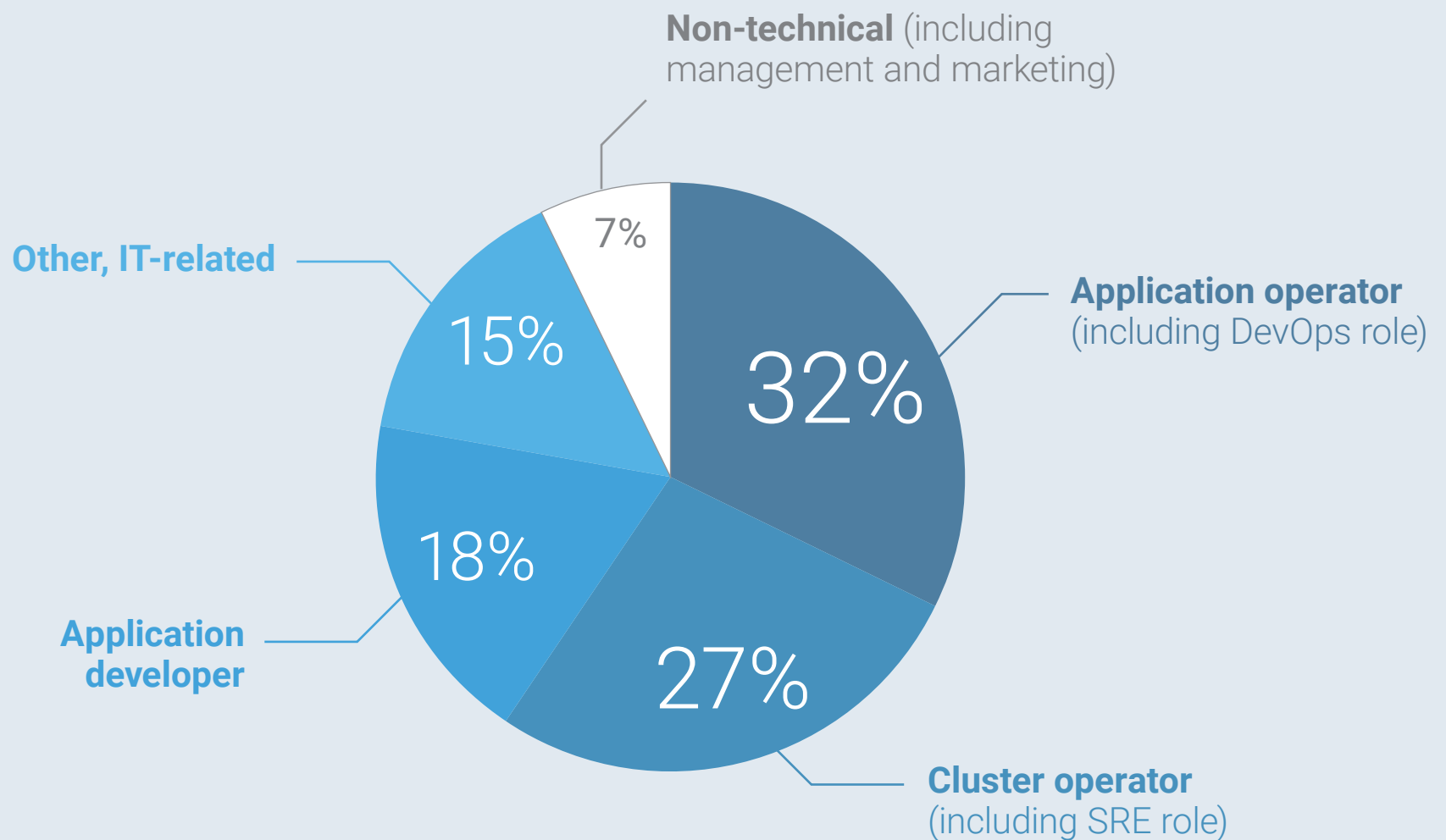
According to [Datadog’s analysis of its own customers](#), large organizations started their container journey earlier than everyone else. It makes sense that big companies are also further along in adopting container orchestration systems that alleviate manual work.

Job Functions of Container Users

We asked all respondents in our survey, regardless of their organization’s state of adoption for Kubernetes, to characterize their job roles with respect to Kubernetes. We gave them a list of four choices — plus Other, IT-related — and asked them to choose the role that best suited their job relationship with the orchestrator, even if it’s in the formative stage.

Continuing what we’ve seen in previous container surveys, a sizable

Job Role



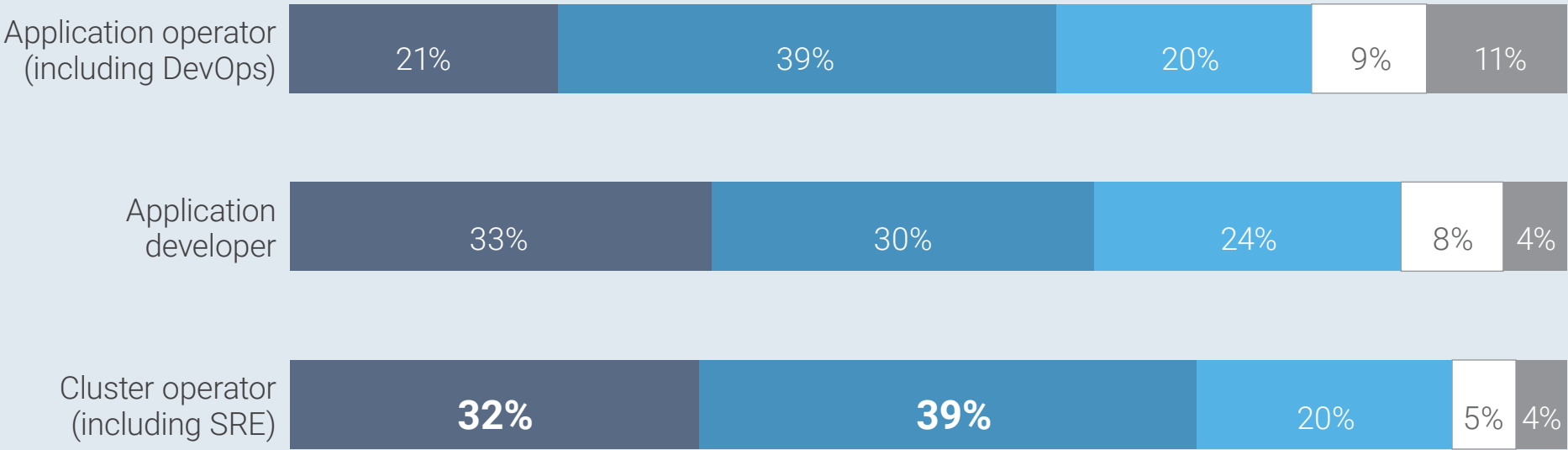
Source: The New Stack 2017 Kubernetes User Experience Survey. Q. What is your primary job role regarding Kubernetes? n=467.

THE NEW STACK

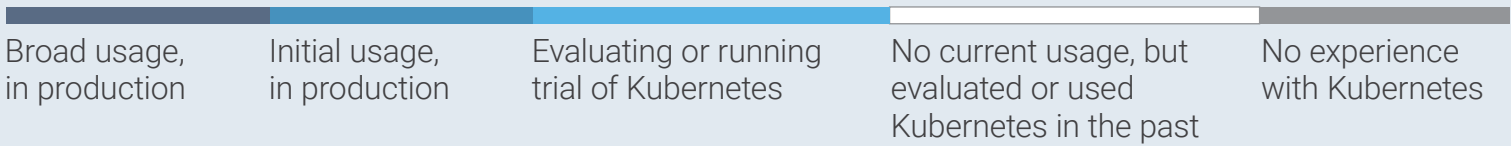
plurality (32 percent) of respondents self-identified as having a DevOps role. For this report, we described someone who manages the operations of applications on Kubernetes as being in a DevOps role. Another 27 percent care about Kubernetes from a cluster operator or site reliability engineering (SRE) perspective. Although the container phenomenon started among application developers, only 18 percent of respondents chose that as their Kubernetes-related job role.

Among cluster operators and those with SRE roles, 71 percent were likely to work in organizations where Kubernetes is part of production — the largest plurality of job roles in our survey. This figure drops to 60 percent among those with an application operator role. Cluster operators may be more likely to be system managers than developers, which may explain why they care more about Kubernetes' management, and how it integrates with other systems.

Kubernetes Implementation by Job Role



Status of Kubernetes Usage



Source: The New Stack 2017 Kubernetes User Experience Survey.
Q. What is the status of Kubernetes usage at your enterprise or organization? Q. What is your primary job role regarding Kubernetes?
Application operator, n=148; Application developer, n=86; Cluster operator, n=127.

THE NEW STACK

Application operators and those with DevOps roles are not necessarily less interested in Kubernetes, but instead may be part of a second wave of interest among people whose organizations have yet to adopt Kubernetes in production.

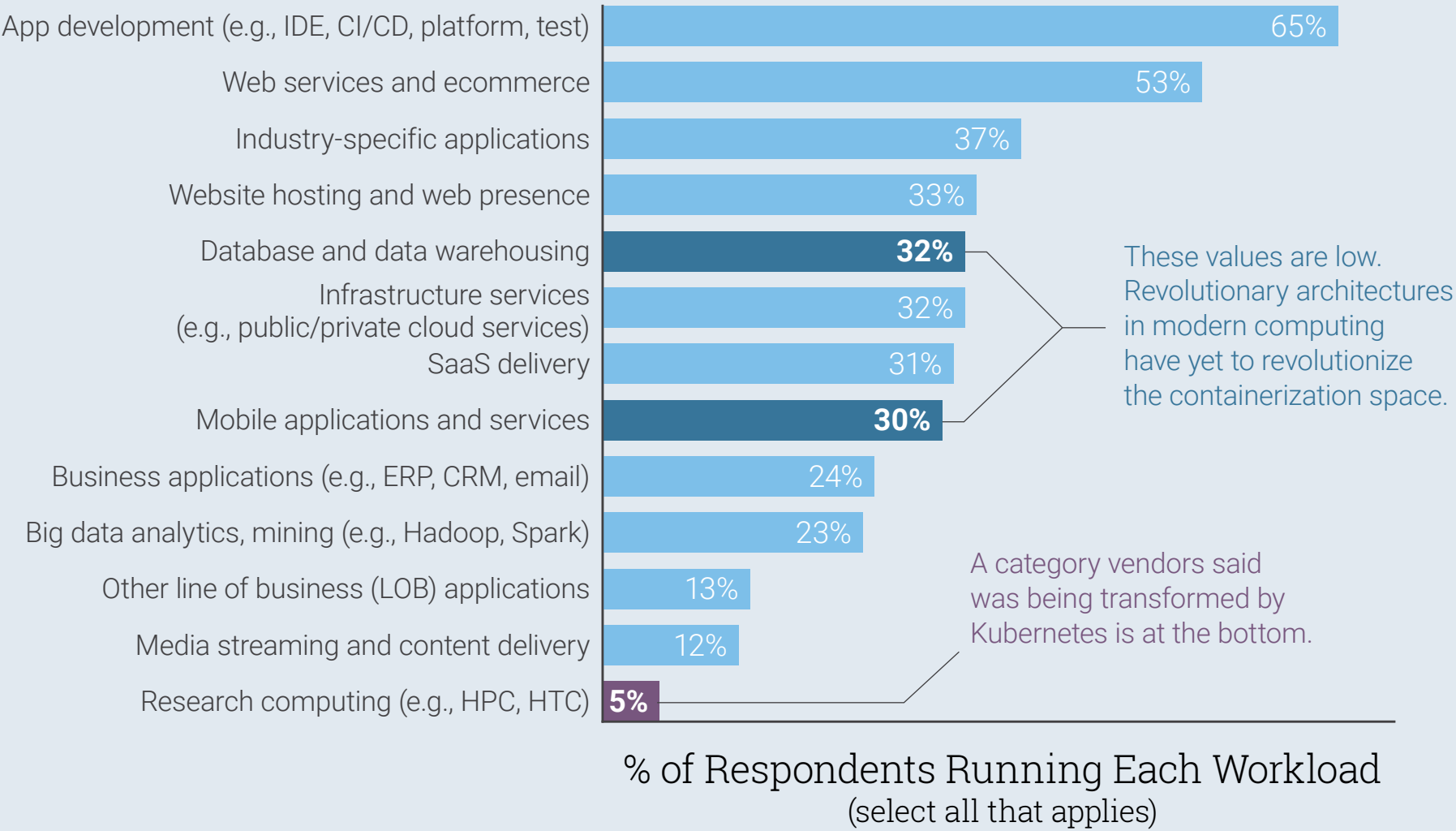
How Kubernetes is Used in Production

For about two-thirds of organizations, Kubernetes — a platform said to be the product of a developer-led revolution — stages applications for use by developers. Although developers are not the majority of our respondents, they’re the ones behind the original Docker revolution, and the ones benefitting first from more efficient deployment pipelines.

Types of Orchestrated Workloads

We gave the subset of respondents whose organizations have deployed applications to Kubernetes in production a list of 13 categories for those

Workloads Running on Kubernetes



Source: The New Stack 2017 Kubernetes User Experience Survey.
Q. What types of workloads does your enterprise or organization run on Kubernetes? n=235.

THE NEW STACK

applications’ workloads, and asked them to choose any and all that applied to their own production workloads.

About two-thirds of Kubernetes production implementations (65 percent) are running applications dedicated to the development process itself. This figure is further evidence that developers are the drivers behind the architecture of orchestrated, distributed systems. Although some e-commerce apps rely on state (for instance, persistent data from databases) to support transactions, web services that don’t rely on state were some of the earliest application types to both be containerized — and, for that matter, to move to a microservices architecture. Not surprisingly, at 53 percent, web services and ecommerce are the second most likely workloads being run on Kubernetes.

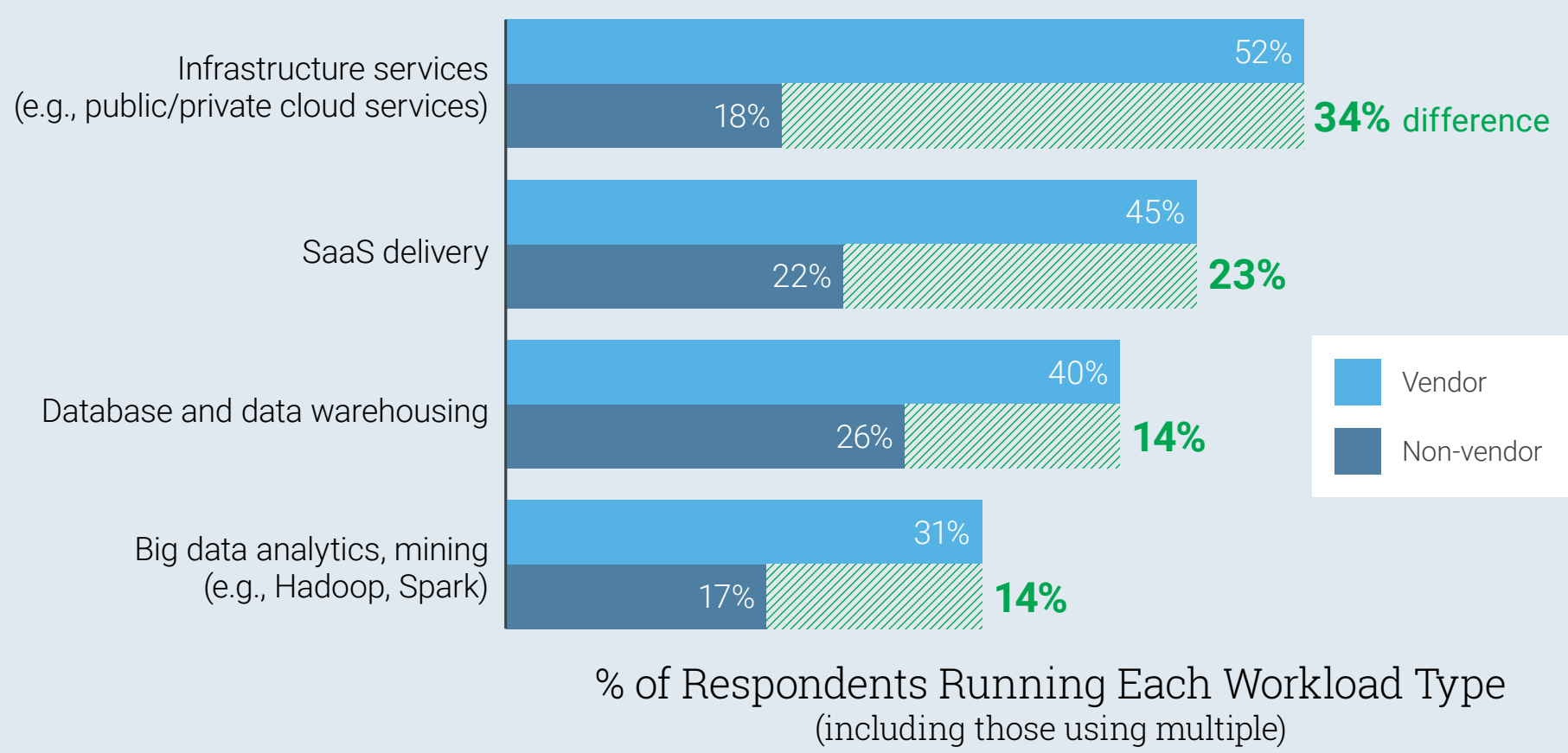
Just fewer than one-third of respondents (32 percent) say their organizations are running databases or data warehousing services on

Kubernetes, and even fewer (30 percent) are running mobile app servers and mobile services. These are far from negligible numbers, but still their low values are indicators that two other so-called revolutionary architectures in modern computing — big data and mobile — have yet to revolutionize the containerization space. Surprisingly, only five percent of respondents said their Kubernetes-based workloads are based around research jobs such as high-performance computing. This is a category that vendors said was being transformed by Kubernetes early in its history.

Use Cases for Vendors

We gave both vendors and end-users four broad categories of services, and asked them to choose any and all that applied to their Kubernetes deployments. In contrast with end users, vendors were more likely to run workloads related to provisioning cloud infrastructure and basic services such as databases. Some non-vendor respondents may receive these

Vendors More Likely to Run Certain Types of Workloads on Kubernetes

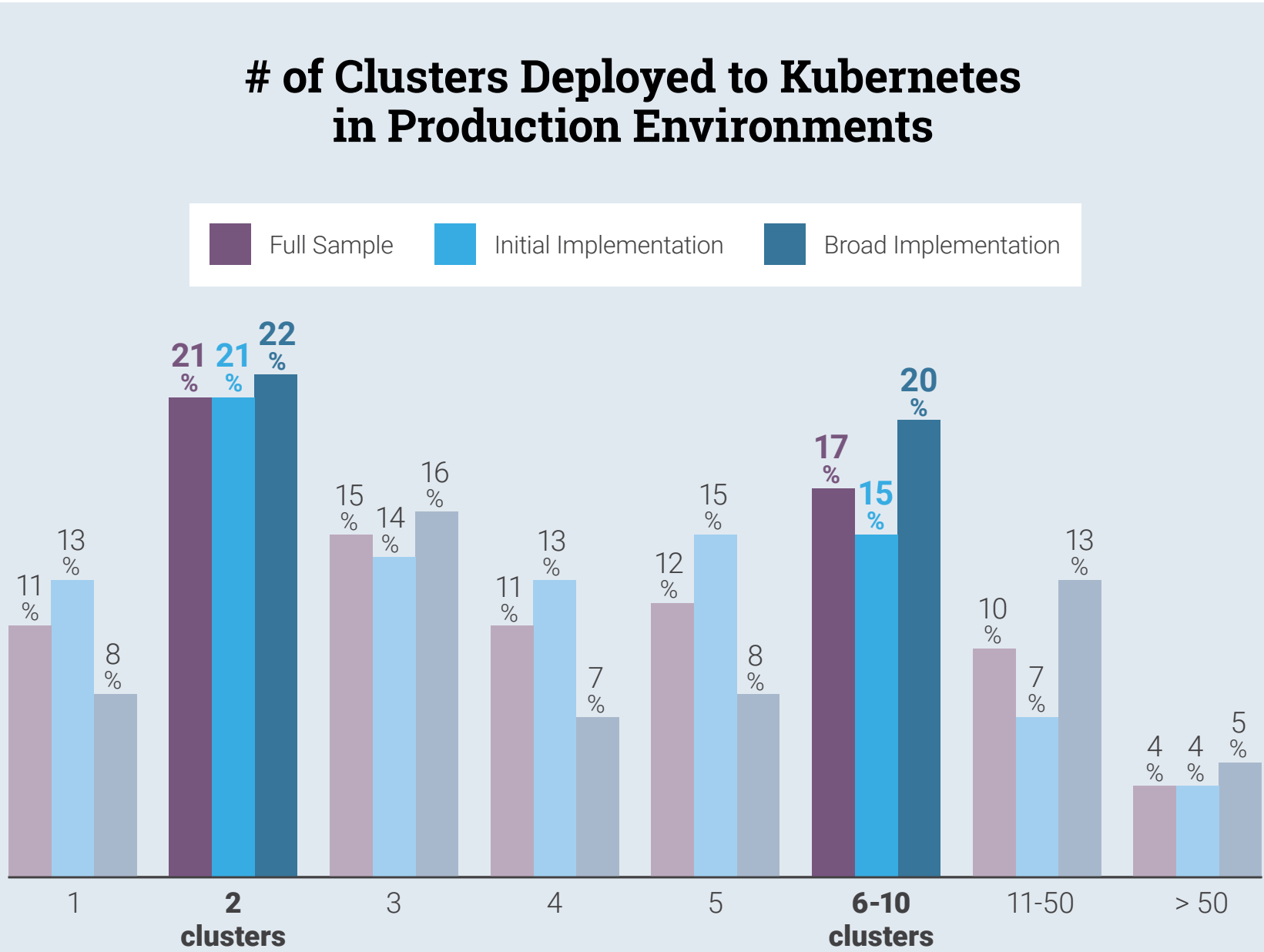


types of services from an external provider that is actually running Kubernetes.

Vendors were more likely to say they were running stateful workloads (the traditional variety) than stateless services. Some 40 percent of vendors responding said they run databases and data warehousing operations on Kubernetes, compared with only 26 percent of non-vendors. Similarly, 31 percent of vendors run big data analytics on Kubernetes, compared with 17 percent of non-vendors. These stark differences are clear indicators that the requirements of hyperscalers and software service providers differ from those of commercial enterprises.

Size and Breadth of Deployments

We asked respondents to quantify the clusters in their organizations' current Kubernetes deployments, and then grouped their answers into eight buckets.



Source: The New Stack 2017 Kubernetes User Experience Survey.
Q. How many clusters are deployed? Broad Kubernetes Implementation, n=98, Initial Implementation, n=131.

Among all respondents, some 21 percent said their organizations deploy two Kubernetes clusters, and that figure stays solid among those who have just begun their Kubernetes involvement (21 percent) and those who are well along in the adoption process (22 percent).

The second peak falls in the six to 10 cluster range. Once the number of clusters reaches double digits, this second “peak” becomes more pronounced. The 17 percent figure for six to 10 clusters represents an average between the 20 percent of broad implementers in that range, and the 15 percent of initial implementers. This should give you a general idea of the steady pace with which clusters are being added to maturing Kubernetes deployments.

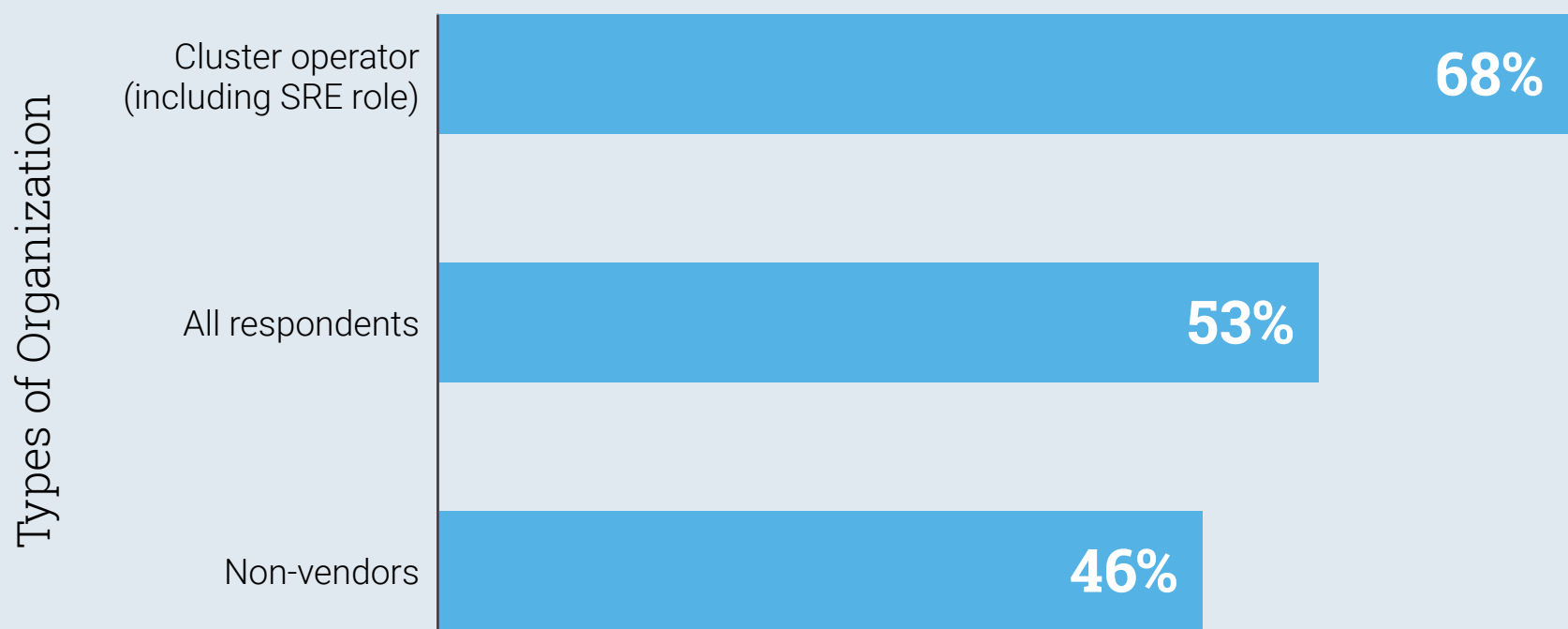
Broad implementations are indeed farther along, as some 38 percent of respondents are managing more than five clusters, compared to only 26 percent of those still in the initial phase. In other technology markets, large enterprises often have larger-scale deployments, but that’s not yet the case among our group of participants.

Based on these responses, we can estimate the average number of clusters deployed on Kubernetes for all implementers at 23.

Multiple Physical Sites

As the scope of workloads being containerized increases, the number of clusters will probably mirror that growth. A roadblock to managing that growth may be the ability to run clusters across multiple clouds and multiple sites. That obstacle can be overcome by technology advances. Another possible roadblock is that companies build out the capability to manage large clusters, but their internal developers do not rapidly migrate workloads over to containers. In both situations, cluster operators will be stuck in the uncomfortable position of managing underutilized infrastructure.

Deployments Spanning Multiple Data Centers



% of Respondents Whose Organization
Deploys Kubernetes in Multiple Data Centers

Source: The New Stack 2017 Kubernetes User Experience Survey.
Q. Does your deployment span across multiple data centers? Full Sample, n=232; Cluster Operators; n=81; Non-vendors, 139.

THE NEW STACK

We asked all survey participants whether their organizations’ Kubernetes deployments spanned multiple physical data centers, to determine whether virtual data centers were indeed expanding beyond the boundaries of physical ones. Slightly more than half of deployments among our respondents (53 percent) do span multiple data centers. Companies that do not provide cloud or software services were less likely to have multi-site deployments (46 percent), but cluster operators were much more likely (68 percent).

A full 97 percent of respondents’ companies that operate Kubernetes across multiple data centers also run more than one cluster. Completely federating the components of a virtual data center across multiple clouds may yet happen, but it hasn’t happened yet.

Resources Usage with Kubernetes

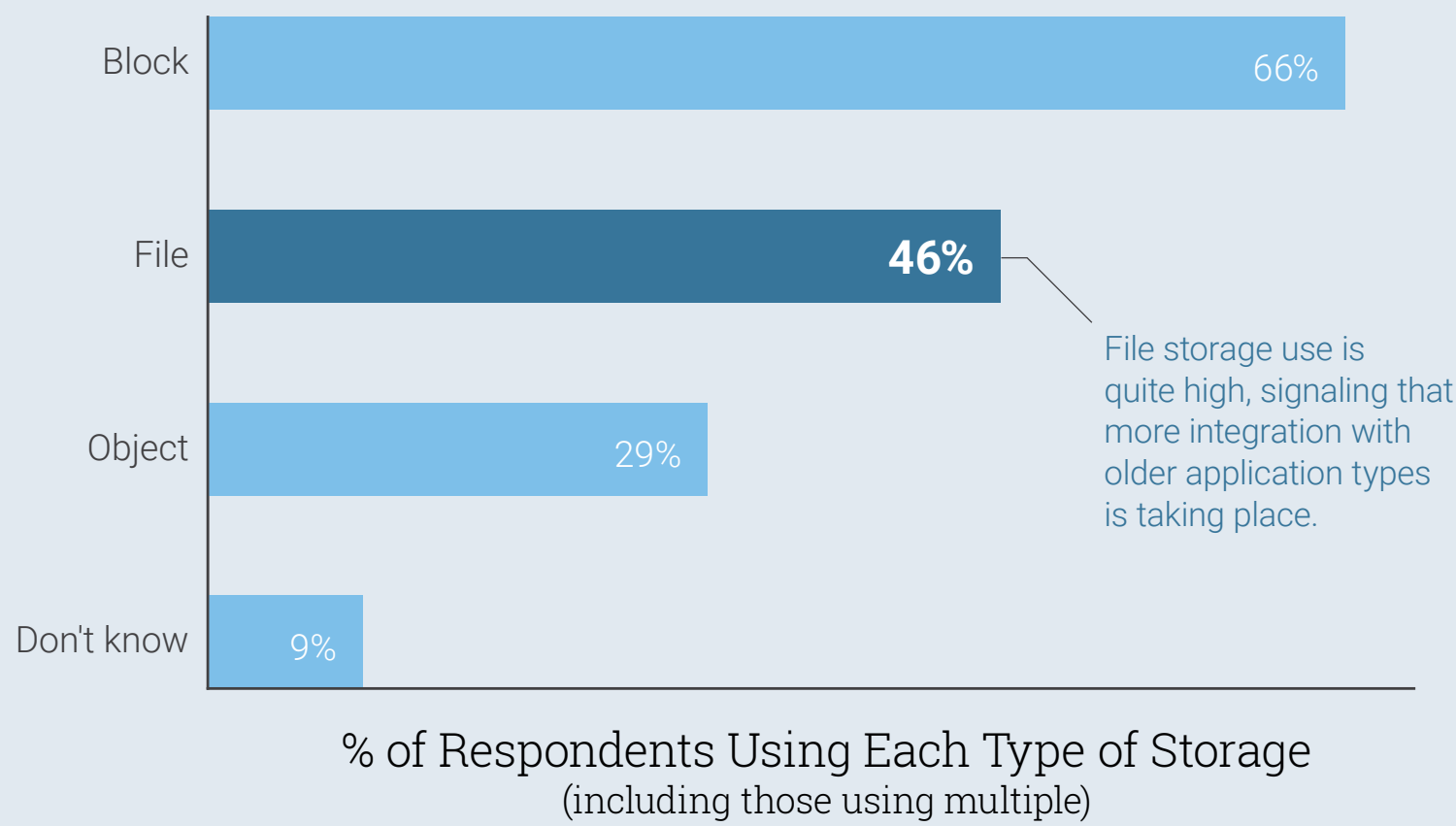
Whether organizations stage their applications mostly in the public cloud or mostly on-premises, clear favorites are emerging for the classes of resources they use with Kubernetes. Adoption patterns for storage, networking and monitoring can inform your own decisions about which tools you use, or which ones you support in your own products.

Storage Systems

The types of logical storage structures used in today’s Kubernetes deployments offer some deeper revelations into the nature of workloads being deployed. Block storage is king, having been cited by two-thirds (66 percent) of our respondents as being involved with their Kubernetes implementations.

Few deployments are relegated to only one type of logical storage, so it is

Types of Storage Used With Kubernetes



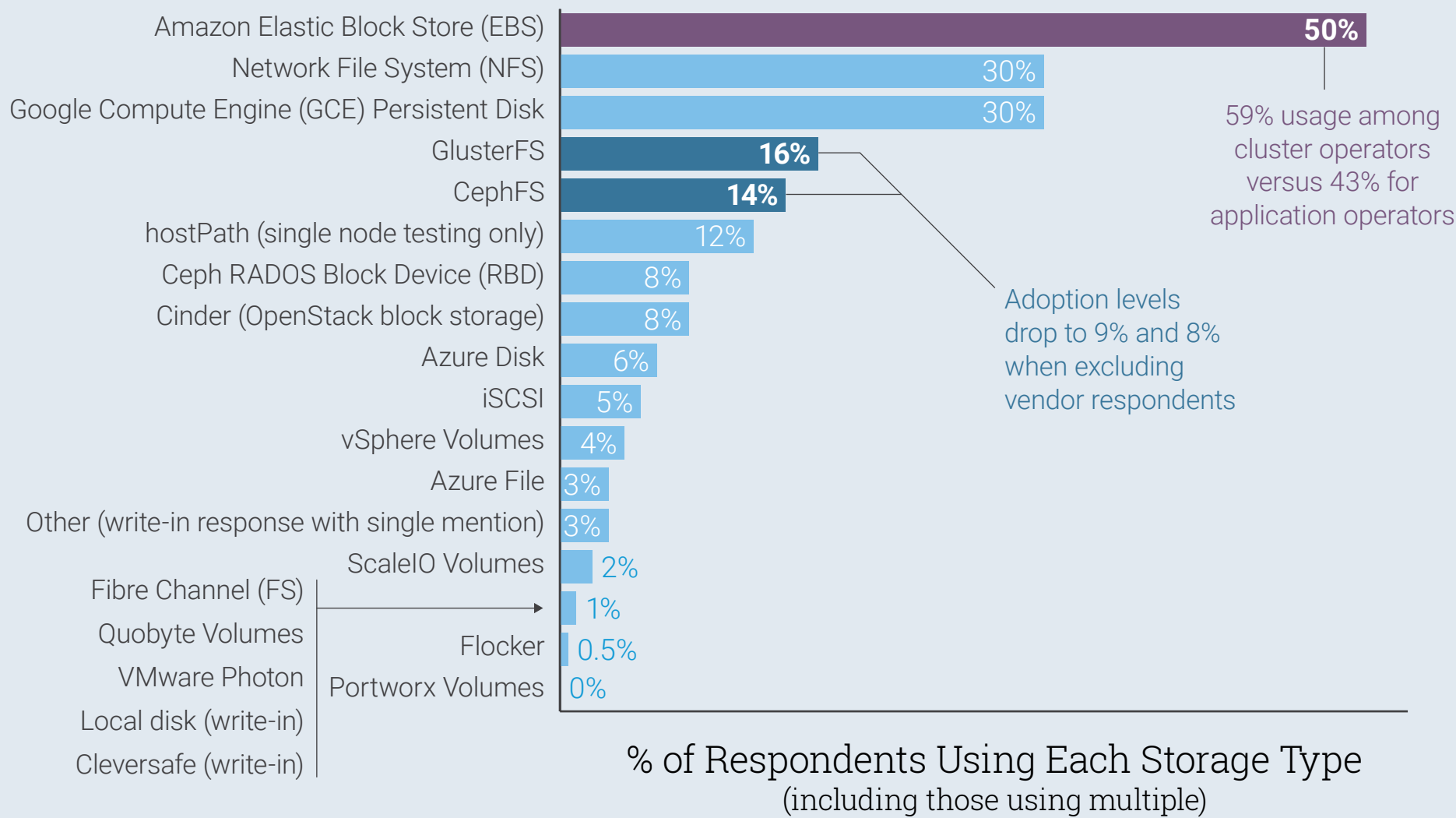
telling that just fewer than half of respondents (46 percent) cited file storage at the type they’re using. Newer, cloud-native applications with microservices architectures, and that utilize databases or data structures, typically don’t need a file system because they are not interacting with data through an operating system. A 46 percent figure is quite high, signaling that more integration with older application types is taking place.

Object storage is used by 29 percent of respondents, which is relatively high compared with adoption rates for object storage that we’ve seen in the past. Since object storage is scalable, developers working on distributed systems likely have experience with it already. In addition, object storage is often used to deliver static content for websites, which is also a common type of workload for Kubernetes.

Providers of Logical Storage Services

We gave respondents a list of specific logical storage system brands,

Specific Storage Used With Kubernetes



projects and technologies, and asked them to cite any and all that their Kubernetes deployments utilize. We gave them an opportunity to write in alternatives that we did not list. Some of these are commercial products or services, while others are open source projects.

Partly due to its strong position in the cloud market, Amazon Elastic Block Storage was cited by half (a full 50 percent) of our survey respondents among all Kubernetes implementations. Admittedly, Google's position of respect among early Kubernetes adopters may have played a greater persuasive role, than its own market share in the cloud space, driving the high percentage of respondents (30 percent) who cited GCE Persistent Disk. So these results are not an indicator that, for instance, Google has three-fifths of Amazon's share.

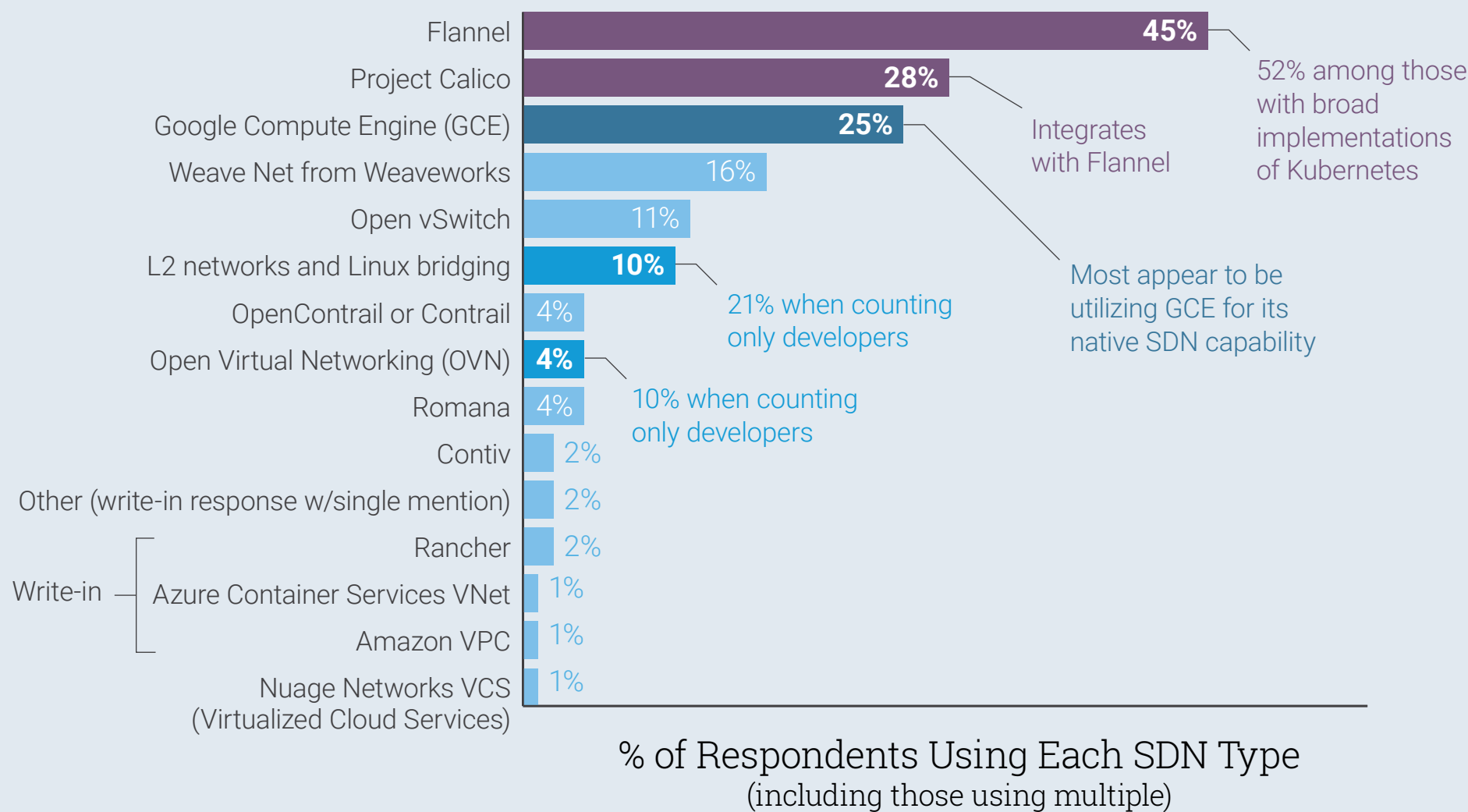
Another indicator of this probable tilt in Google's favor among our group is the fact that only seven percent of respondents cited using Azure-branded logical storage. So Microsoft's cloud market share was evidently under-represented here. The likely reason is that Kubernetes only became an official Microsoft offering in February 2017. As its Azure Stack becomes more widely available, its customers should have more storage options available to them besides just Azure. As a result, we expect Azure developers will be more likely than customers of other clouds to utilize storage they purchased directly from a source other than a major cloud service provider.

Since cluster operators often belong to IT operations teams, it would logically follow that they would be more likely to use internal storage resources. At least among our survey participants, this isn't the case. Some 59 percent of cluster operators use AWS storage, but only 43 percent of application operators (DevOps) do the same. Perhaps using the AWS cloud lets SREs focus on infrastructure monitoring, rather than suffer the constant headaches associated with managing hardware.

Many respondents said they used a specific file-based storage system, with NFS being cited most often (30 percent), followed by GlusterFS (16 percent) and CephFS (14 percent). However, the latter two were cited more often by vendors. Omitting these vendors, we're left with only nine percent of the remainder using GlusterFS, and eight percent using CephFS. Quite likely, many of these vendor respondents are employed by Red Hat (which utilizes all three of these systems) or one of its many partners. This would be in accordance with a trend we've seen before where OpenStack vendors have adopted Red Hat-backed storage standards more than others.

Container-specific storage solutions were seldom cited. Perhaps as more persistent workloads move to containers, demand for specialized storage options may increase. Another possibility has to do with convenience. Users may prefer the storage they already know, or the storage that's easiest for them to access. The latter explanation would not portend well for the startups that follow in the footsteps of now-defunct ClusterHQ.

Software-Defined Networking Used With Kubernetes



Source: The New Stack 2017 Kubernetes User Experience Survey.
Q. What is being used for software-defined networking (SDN) in your Kubernetes implementations? Select all that apply. n=197.

Software-Defined Networking

Next, we gave respondents a list of software-defined networking products, projects and technologies, and asked them to cite any and all that applied to their Kubernetes deployments. We gave them an opportunity to write in alternative responses we didn't list.

Overwhelmingly, Flannel was the most cited SDN component with 45 percent of respondents citing, followed by Project Calico at 28 percent, and 25 percent citing GCE's native SDN tools.

Digging deeper into these numbers, we found that CoreOS' Flannel was more likely to be used by those with broad implementations of Kubernetes (52 percent of respondents) as opposed to those with initial implementations (39 percent). The Flannel system is a network fabric explicitly for containers, that relies upon a small agent called flanneld installed on each host. Flannel has garnered a reputation of "just working," so there's no evidence on the horizon yet that its top position is in any jeopardy.

Tigera, the company behind Project Calico, has been working on integrations with Flannel, so it isn't surprising that 46 percent of those who cited Calico also cited Flannel. Project Calico has also seen recent success among cluster operators. Some 40 percent of the 40 cluster operators who said their Kubernetes implementations are in the initial stages say they are using Project Calico. Whether Tigera can retain these users through their journey to broader adoption is an open question.

Among the 25 percent of respondents citing Google Compute Engine, most appear to be utilizing its native SDN capability. One respondent who works at a company with between 2 and 100 employees said GCE has "superior container-level networking ... compared to experience with Mesos and Weave overlay network." If others agree, then users won't have

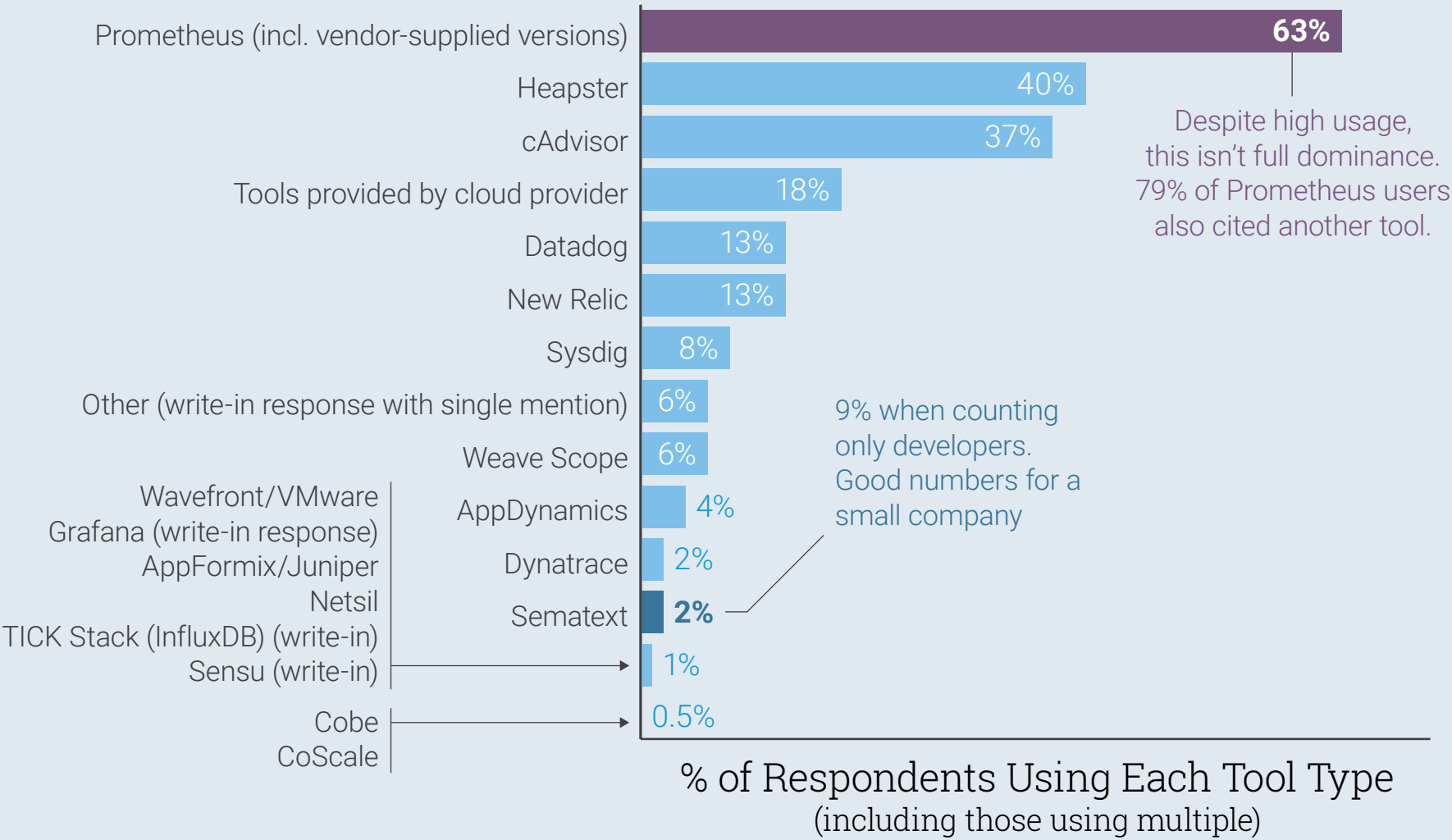
to cobble together an SDN solution if they use Google’s services.

Application developers have varying needs, so it’s not surprising that their choices for SDN approaches were all over the map. Developers citing use of L2 networks and Linux bridging jumped to 21 percent of respondents, compared with 10 percent for the full sample. Use of Open Virtual Networking (OVN) was also higher among developers compared to all respondents (10 percent versus four percent). Naturally, this shows that tools made by developers will be more familiar to developers. But it also suggests that non-developers may not be familiar with the SDN tools their developer counterparts are using.

Cluster Monitoring

Organizations’ choices of monitoring tools for use with Kubernetes speaks volumes about their application deployment strategy. As you’ll see elsewhere in this book, an organization may prioritize monitoring its

Tools/Services Used to Monitor Kubernetes Clusters



Source: The New Stack 2017 Kubernetes User Experience Survey.
Q. What tools, products and services are being used to monitor Kubernetes clusters? n=208.

infrastructure over its applications, or vice versa. Which target an organization chooses also speaks to who does the monitoring: developers or operators.

We gave Kubernetes users a long list of systems monitoring tools of various types, and asked them to choose any and all that were being used with their deployments. We gave respondents an opportunity to write in alternatives that we had not listed.

Among those responding, some 63 percent monitor their clusters using Prometheus, the services monitoring components whose project, like Kubernetes, is shepherded by the Cloud Native Computing Foundation. But this doesn't mean Prometheus has already achieved dominance over other tools and strategies, as some 79 percent of those who cited Prometheus also cited another tool. The modern data center monitoring strategy typically involves either acquiring an amalgam of tools intentionally, or piecing them together from what the data center has already used for years. In fact, the average organization we've polled uses 2.2 technologies to monitor its clusters.

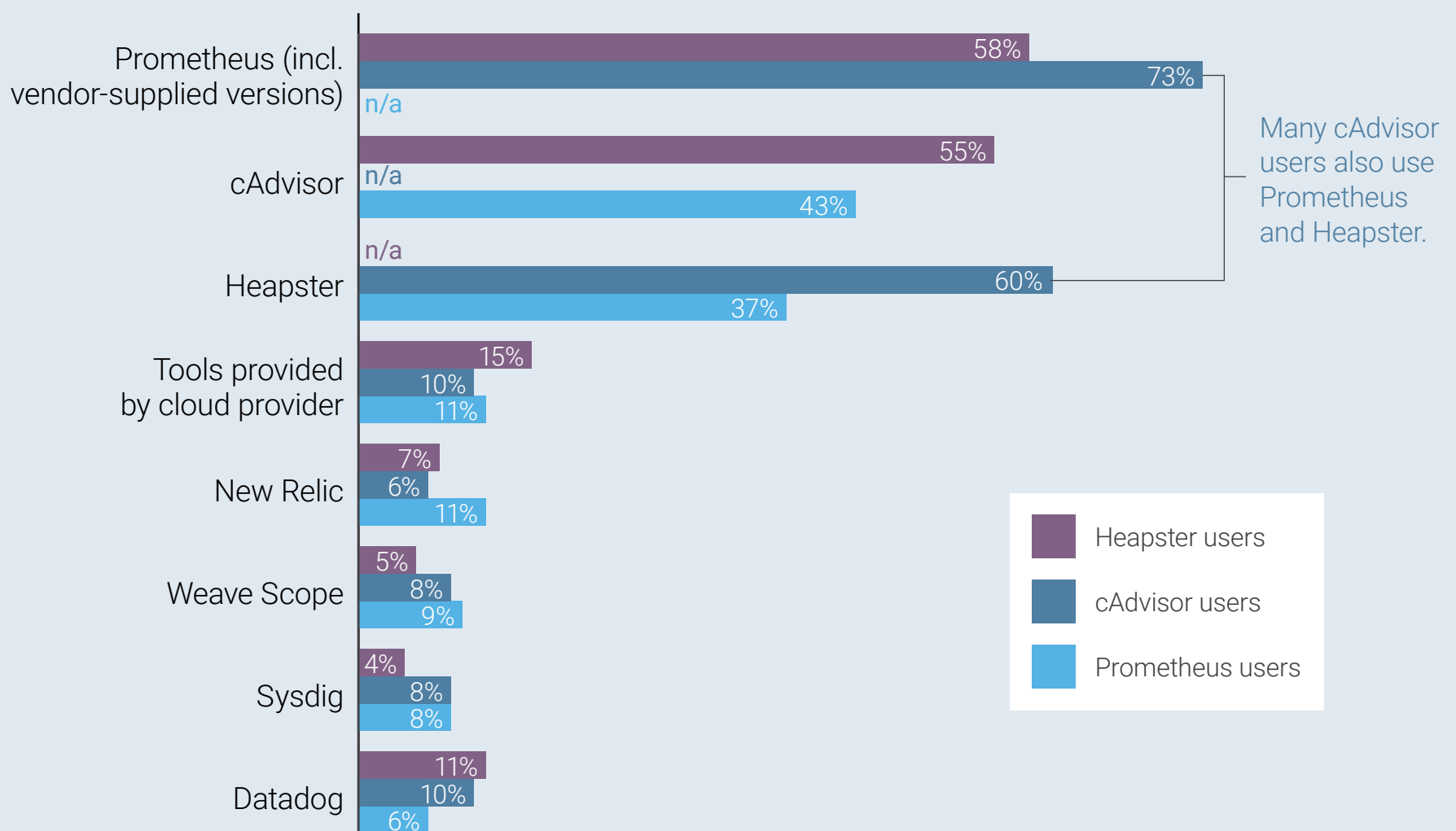
Also widely cited as having been adopted are Heapster (40 percent), which monitors resource usage events for Kubernetes specifically; and cAdvisor (37 percent), a resource usage monitor created for Docker containers.

Sematext has more visibility among application developers. Some nine percent of developers responding say they use its Kubernetes Agent, which is four times greater than the full sample — a good number for a small company.

Pairing Resource Monitors Together

Prometheus' project leaders could strengthen its position in the data center by continuing the strategy of integration with other tools. The

Kubernetes Monitoring Tools/Services Used With Each Other



Source: The New Stack 2017 Kubernetes User Experience Survey. Q. What tools, products and services are being used to monitor Kubernetes clusters? Prometheus Users, n=131; cAdvisor Users, n=77; Heapster Users, n=84.

THE NEW STACK

prospects for this are especially strong with cAdvisor, 73 percent of whose users also use Prometheus, while some 60 percent of cAdvisor users also have Heapster. Since cAdvisor provides data inputs to other tools, its integration-by-design strategy makes perfect sense.

DevOps professionals in our survey revealed a more hands-off approach to managing Kubernetes, stating they were less likely to use Heapster and cAdvisor. However, they were also less likely to use tools provided by a cloud provider (six percent versus 18 percent of all respondents). By definition, DevOps should be the convergence point for application and systems monitoring. If indeed such a convergence point exists, then application operators should be paying attention and weighing in on their organizations' monitoring strategies — especially as they're being cobbled together from vendor-based and open source options.

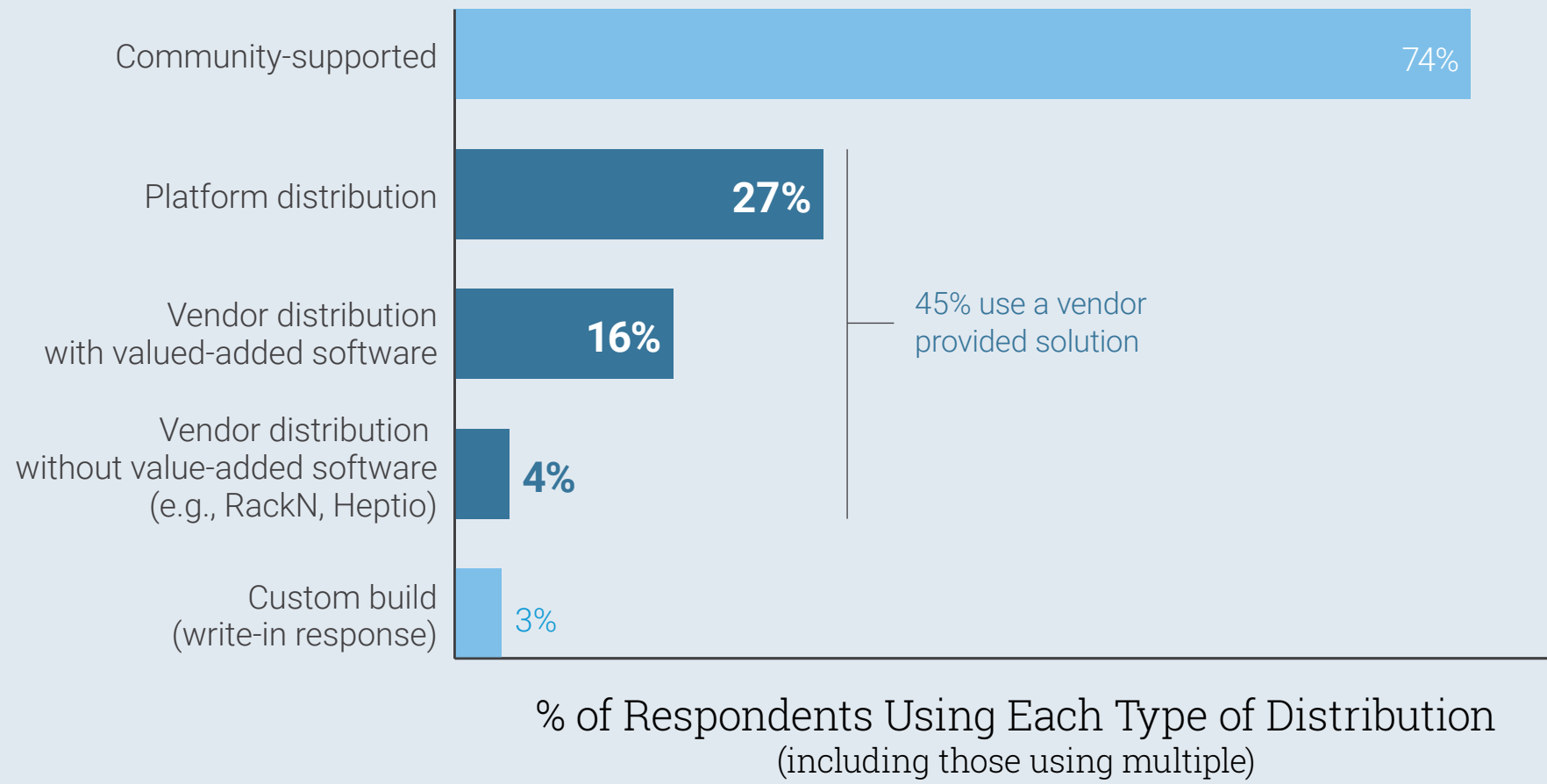
Traditional application performance management (APM) and

infrastructure monitoring vendors are much less likely to be cited by Prometheus users. Datadog monitors Kubernetes clusters for about one non-Prometheus user in four. If Datadog’s customers are satisfied with the monitoring strategy they’ve already employed, they may not end up using Prometheus at all. Absent that, incumbents will have additional opportunities as some users look to full stack monitoring solutions.

Community First, Vendors Later

If you have heard one thing about Kubernetes, it’s that its support community is really strong. One respondent to our survey who works at a mid-tier enterprise put it this way: “The community and involvement of other companies seemed way huge.” Along with its technical merits, the strength of its community is a key reason Kubernetes even had a chance to be adopted so quickly, across such a diverse range of enterprises.

Types of Distributions Used



Source: The New Stack 2017 Kubernetes User Experience Survey.
Q. What types of distributions are being used? (multiple responses allowed) n=173.

Community Options versus Vendor Products

Many observers believe that for Kubernetes to continue its growth trajectory, vendors will need to provide more support and services. Certainly we've already seen a significant uptake of vendor-based options in recent months.

But the strength of the community-supported option remains formidable. For now, 74 percent of total respondents said their organizations use a community-supported distribution of Kubernetes — typically, a version pulled from GitHub. Among this group, only one quarter use an additional, non-community supported distribution.

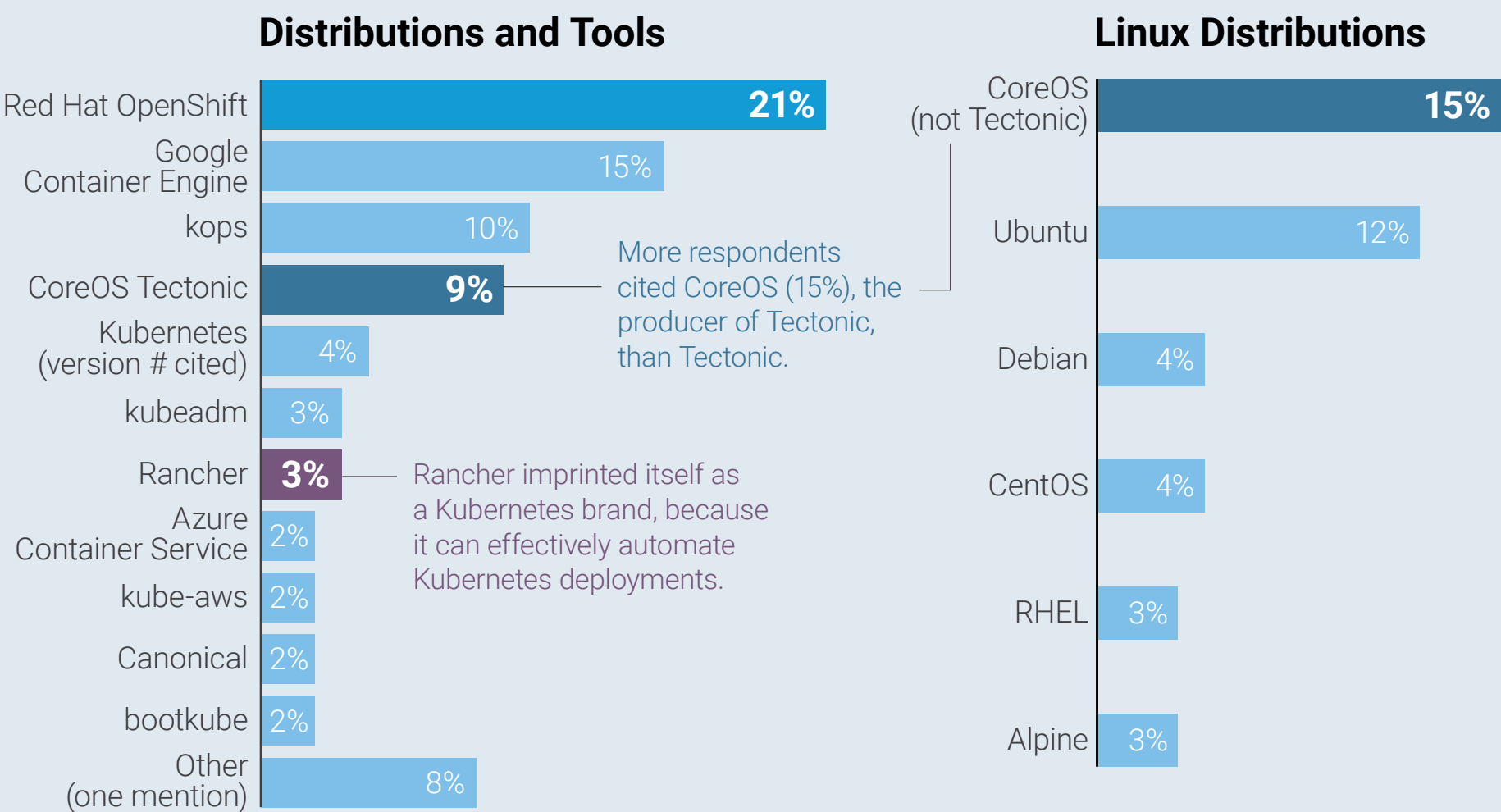
Kubernetes has simply not been around long enough for us to ascertain how quickly organizations transition community-supported Kubernetes to vendor distributions, or even the extent to which that transition is even a priority. However, we do know that a total of 45 percent of respondents used a vendor-offered product of some sort. About half of this group say they use a platform, with another 16 percent saying they use a vendor distribution combined with other valued-added software. A few respondents said they use a vendor distribution that provides Kubernetes support and little else.

Brand Recognition

This next line of inquiry tosses a live grenade in amongst our participants, to see just how they'll react. We asked survey participants to write the one or two names from that list that were the names of their “distribution.” That's vague phrasing, and it's vague on purpose.

Our intent was to determine which brands Kubernetes users associate with Kubernetes. Conceivably, a “distribution” could refer to the vendor who provides the Kubernetes platform, or the maker of the Linux distributed with that platform, or perhaps the cloud service provider on

Brands and Technologies Associated with Kubernetes Implementations



Source: The New Stack 2017 Kubernetes User Experience Survey. Q. What are the names of the main one or two distributions being used? Please include as much information as possible about the vendor, service name and/or version number. n=91. THE NEW STACK

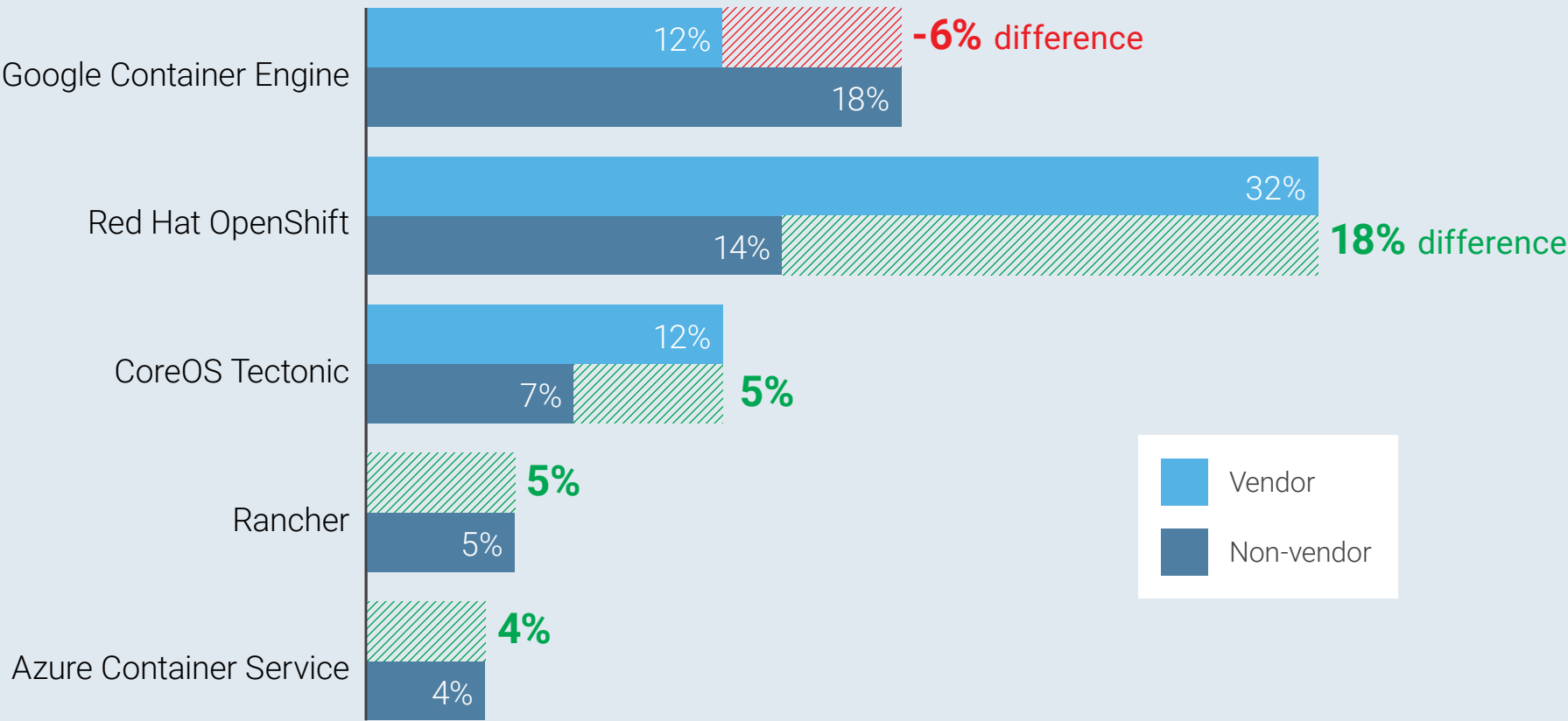
which the orchestrator runs.

Greater than one respondent in five (21 percent) cited Red Hat OpenShift, for the largest plurality of responses to this line of inquiry. But as we discovered when breaking down responses, among those participating who were employed by vendors, a full 32 percent cited OpenShift, compared with only 14 percent who did not work for software vendors.

Breaking down responses further, we determined that only 12 of the 19 respondents who cited OpenShift believed it was a vendor-based distribution with value-added features.

Many respondents named their Linux distribution as their Kubernetes provider. In fact, more respondents cited CoreOS (15 percent) than Tectonic (nine percent), even though CoreOS is the producer of Tectonic. Some 10 percent of respondents mentioned kops, which is actually a provisioning and installation tool for Kubernetes. Respondents who use a

OpenShift's Lead Partly Due to Vendor Participation in the Survey



Source: The New Stack 2017 Kubernetes User Experience Survey. Q. What are the names of the main one or two distributions being used? Please include as much information as possible about the vendor, service name and/or version number. Vendor, n=34; Non-vendor, n=57. WARNING: Small Sample Size.

THE NEW STACK

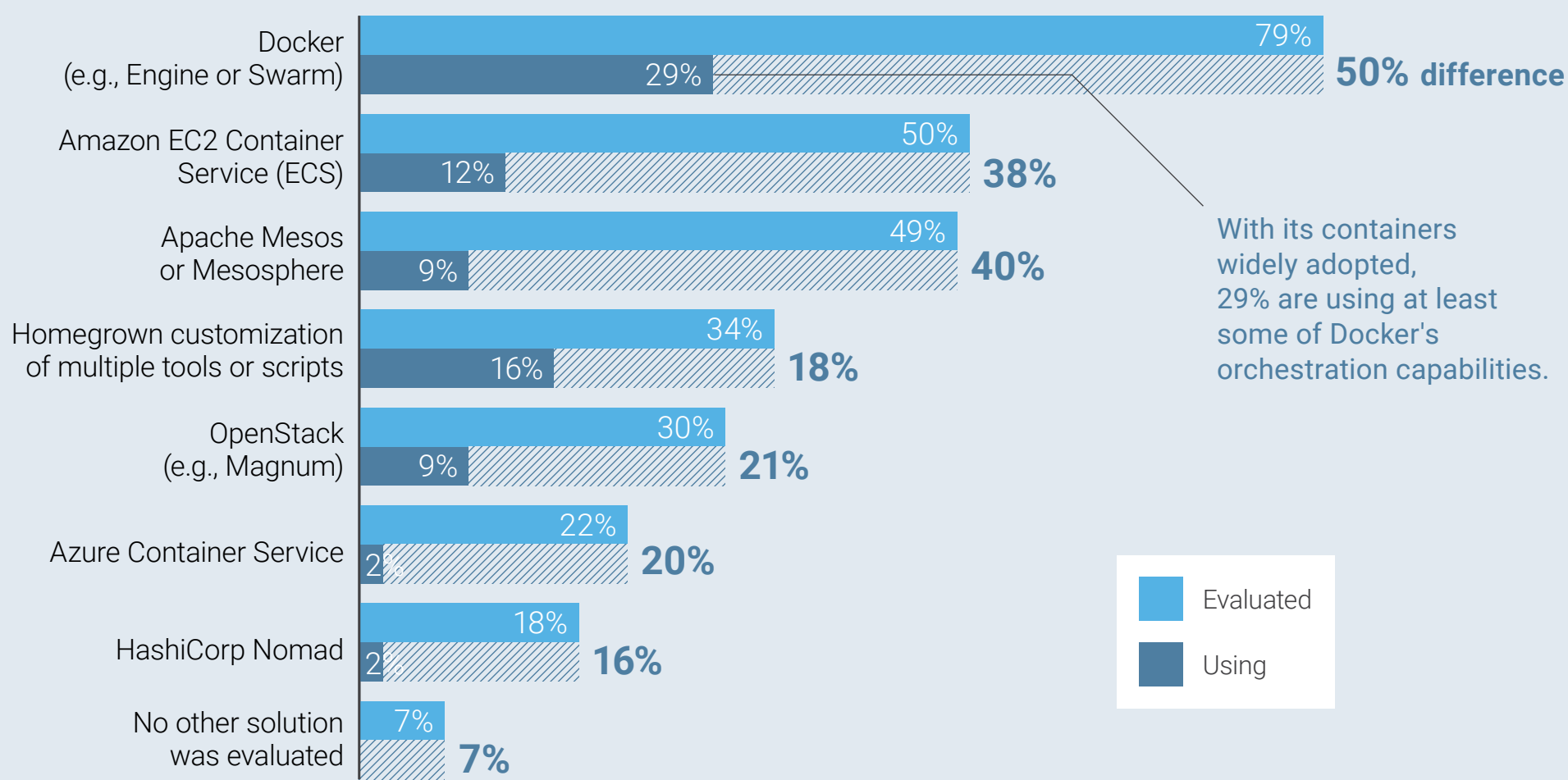
community-supported Kubernetes often look to kops to provide them with the most up-to-date release.

Rancher is an established container management platform, produced by Rancher Labs. Although it does not include Kubernetes, it can automate Kubernetes deployment very effectively. Some three percent of total respondents cited Rancher as one of the Kubernetes “distributions” we asked for. So among these few people, Rancher imprinted itself as an effective Kubernetes brand.

Evaluating Kubernetes and Competitors

People do not choose Kubernetes in a vacuum. A full 93 percent of our survey respondents told us their organizations had evaluated alternatives to Kubernetes before making their choice.

Evaluation and Use of Other Container Orchestration Solutions



Source: TNS 2017 Kubernetes User Experience Survey. What were the other solutions your enterprise or organization evaluated? What, if any, are currently in use? Evaluated n=111, Using n=195. NOTE: It was difficult to differentiate whether write-in responses were for evaluation or use. Rancher received 9 write-ins (3% of evaluated responses). Cloud Foundry received 5 (2%). Heroku and Google App Engine received 2 each.

Openness to Alternatives

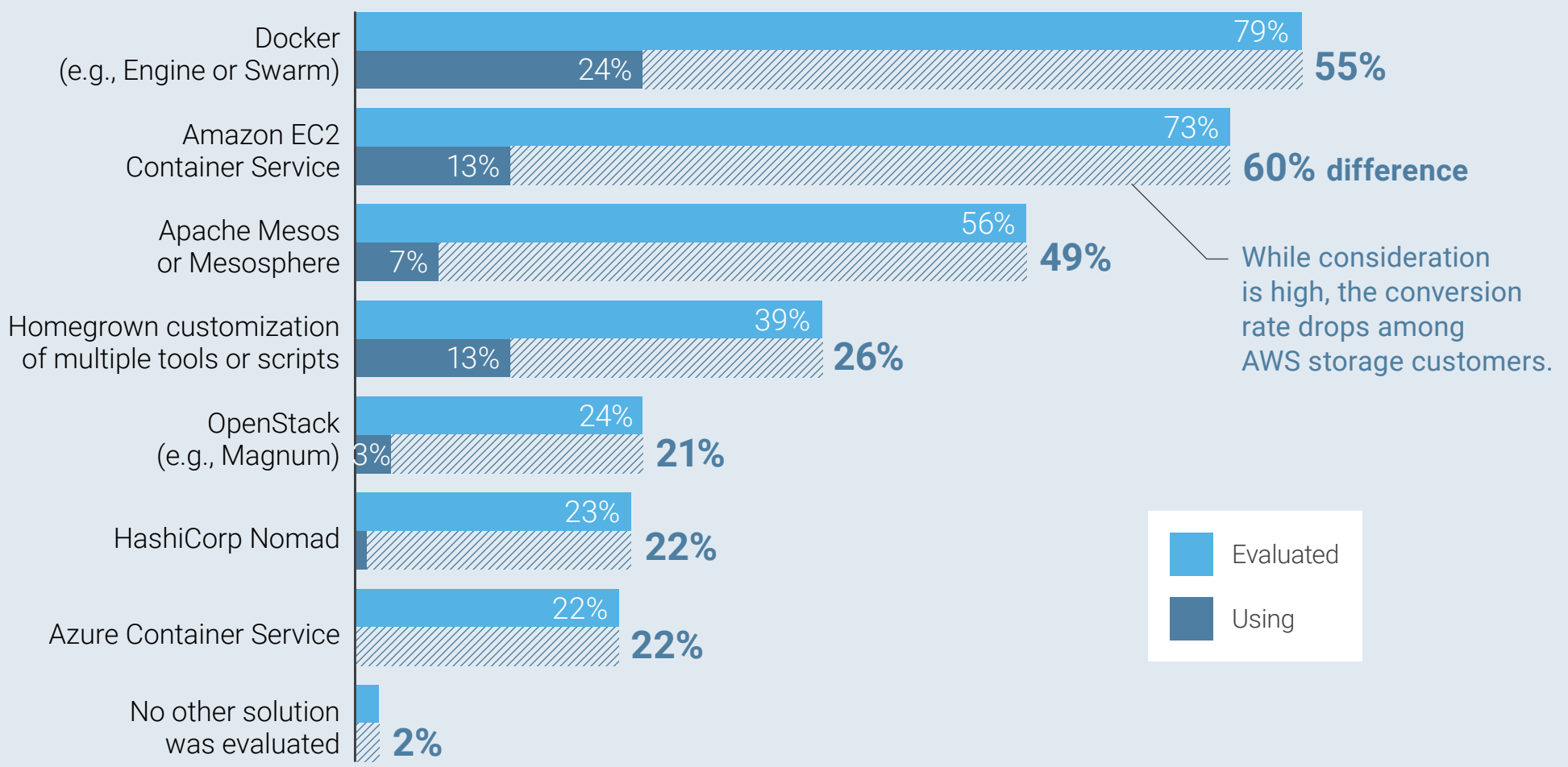
What’s more, a full 45 percent of respondents told us they are simultaneously utilizing other application platforms. Of those that have not adopted Kubernetes, 61 percent said they are using a competitive product.

The buildout of tools for managing these new distributed environments is still shaking out. So there’s still rays of hope for Kubernetes’ competitors. Clearly the market itself lacks definition, and even many Kubernetes users are still thinking about alternatives.

Perspectives on Competitors

We gave survey participants a list of seven container platforms, plus a No Choice option, and asked them to choose any and all that their organizations are presently evaluating, and also that they’re currently using. We saw an interesting set of results, indicated by this chart, among

Evaluation and Use of Other Container Orchestration Solutions by Those Utilizing AWS Storage With Kubernetes



Source: The New Stack 2017 Kubernetes User Experience Survey. What were the other solutions your enterprise or organization evaluated? What, if any, are currently in use? What specific storage volumes does your enterprise or organization use with Kubernetes? n=90.

THE NEW STACK

the subset of respondents who had also indicated they were using Amazon AWS storage services in conjunction with their Kubernetes deployments.

We’ve stated in The New Stack before that the rise of Kubernetes was made possible by the revolution started by Docker. In 2016, Docker Inc. began bundling Swarm orchestration capabilities into Docker Engine. So it’s reasonable to assume that organizations have had ample opportunity to evaluate Swarm.

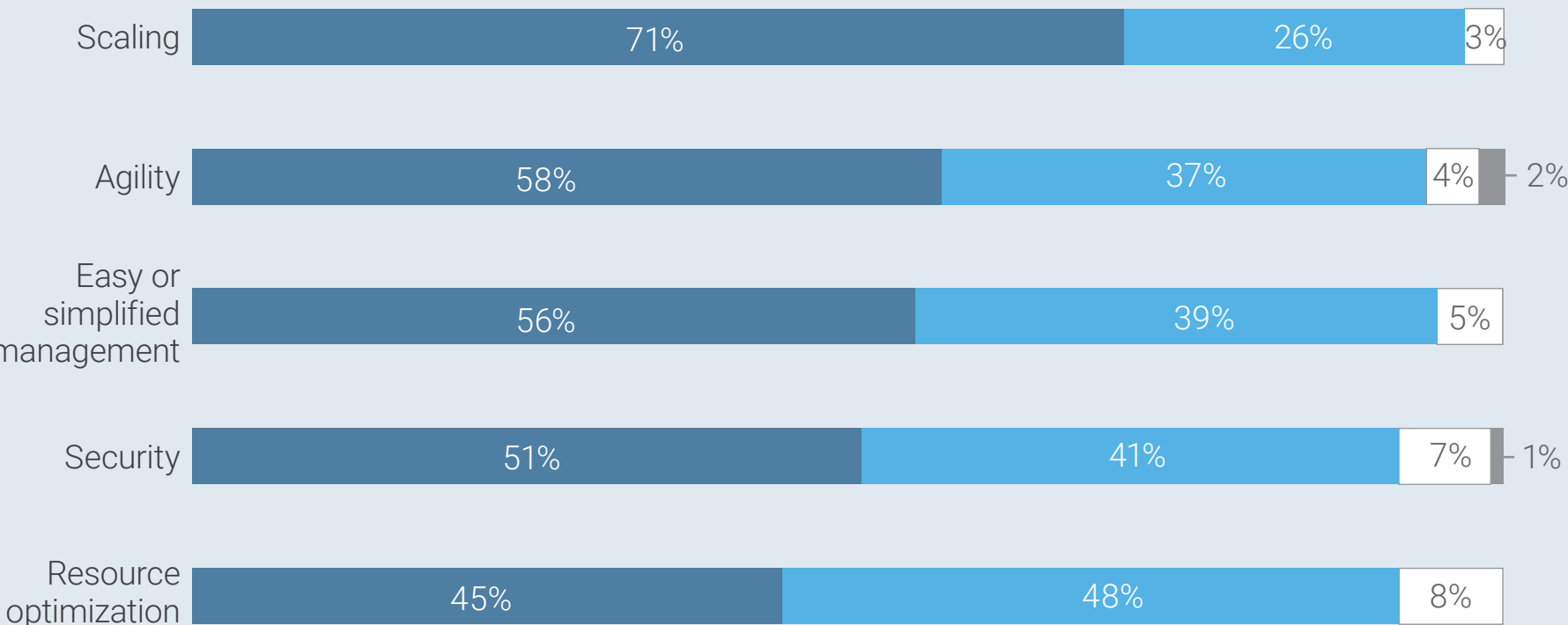
Among the Kubernetes evaluators who responded to our survey, nearly three respondents in four (73 percent) who use AWS storage with Kubernetes gave Amazon EC2 Container Service some consideration. Very few of these people (18 percent) actually ended up using ECS. The high level of consideration is mostly because of AWS’s overall Infrastructure as a Service (IaaS) market share. Usage levels among non-vendor

respondents were slightly lower for both ECS and Mesos/Mesosphere. Here are a few comments from survey participants employed by businesses with 100 employees or fewer, who evaluated these two platforms:

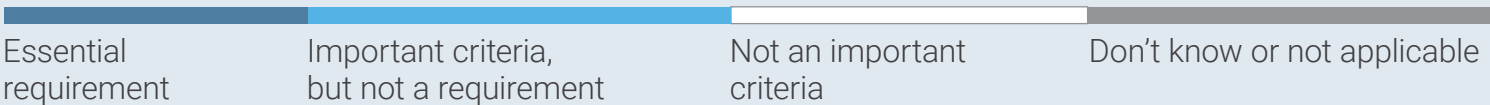
- “It’s hard work with K8s on top of AWS. After a year and a half, some tools are growing up like kube-aws and others like kops still destroying your cluster every time you update the cluster. Choose the right tool. It’s very important and CloudFormation is mandatory if you wanna use K8s in AWS and be safe.”
- “AWS need to integrate K8s as service like GCP; otherwise, we’ll migrate.”
- “Mesos is very painful to use if you want to write your own scheduler: you must compile your scheduler against the same version of libmesos. So you are running, and libmesos doesn’t compile quickly or easily.”

Evaluation Considerations for Container Orchestration

Evaluation Considerations



Importance of Consideration



Source: The New Stack 2017 Kubernetes User Experience Survey. Q. To what degree were the following considered when evaluating Kubernetes and other container orchestration solutions? n~313. Due to rounding, percentages may not always appear to add up to 100%

Decision Criteria

Now that we know which platforms our survey participants were considering, let's look at the criteria respondents were using to evaluate container orchestration offerings. We gave participants a list of five categories for evaluating orchestration platforms, and asked them to rate each one on a four-point scale, ranging from “Essential requirement” to “Not applicable.”

Respondents overwhelmingly cited all five factors as important criteria, with only resource optimization falling below half of responses for essential requirement (45 percent). The ability to scale was rated essential by 71 percent of respondents.

People currently evaluating Kubernetes were more likely than those using it in production to view security as an essential requirement — 65 percent of evaluators, compared to 45 percent of users. Here are some revealing comments from two people in the users' column:

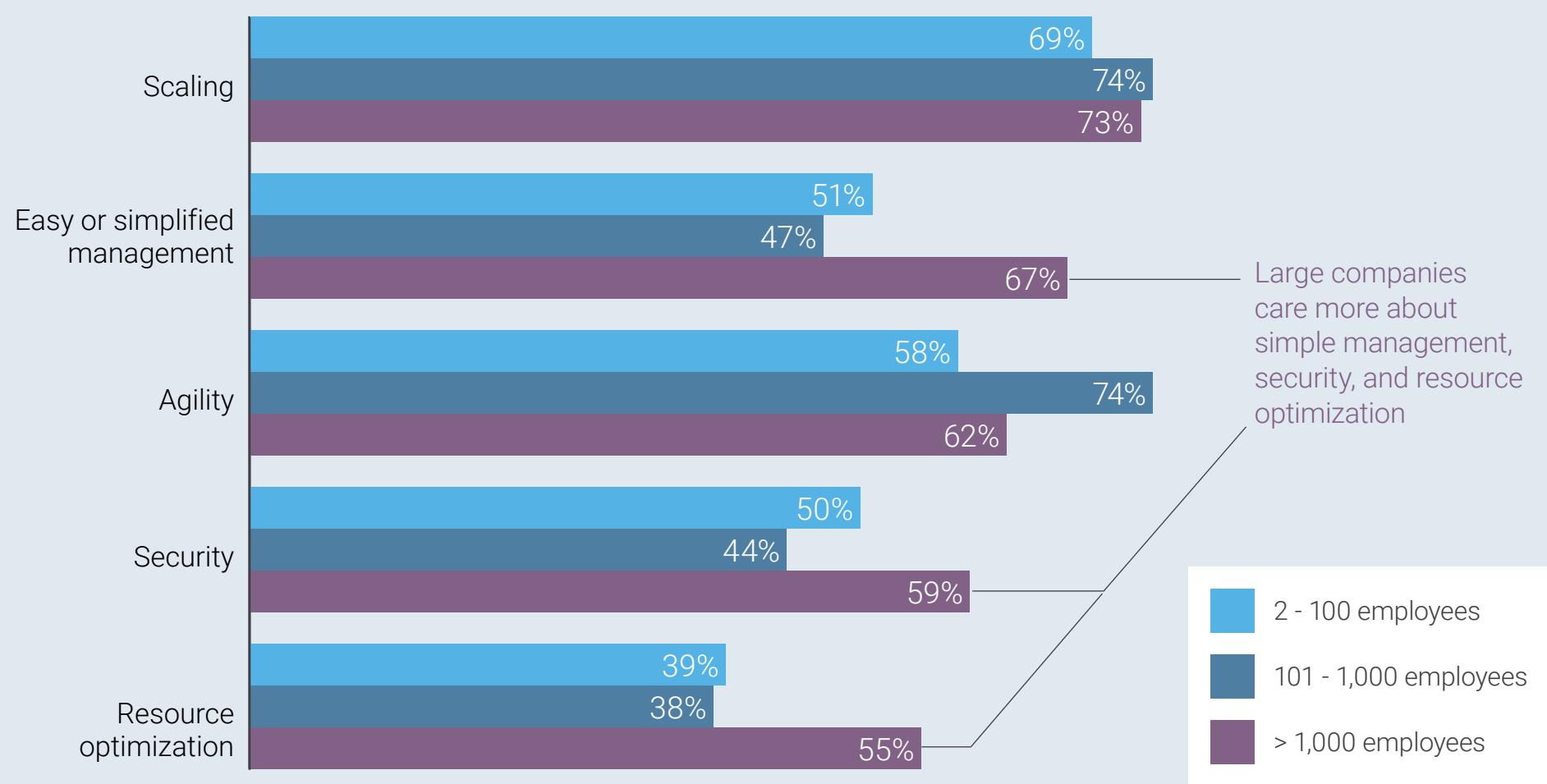
- “Need a secure — really secure, not what is currently available or proposed — method to access and manage data both in the clustered application, between clustered applications and beyond.” [Software company, 2 - 100 employees]
- “We run a fairly large Web shop (100M Euro in revenue per year). Our application is/was monolithic and we needed a way to scale more on demand. K8s provides the basis for scalability on-demand.” [101 - 1,000 employees]

Breaking Down Decision Criteria

When we break down responses to the decision-making criteria question by the employee count of respondents' employers, we find some surprisingly stark differences in their patterns. Large companies (with

Considered Essential When Evaluating Container Orchestration

(by Number of Employees)



Source: The New Stack 2017 Kubernetes User Experience Survey.
Q. To what degree were the following considered when evaluating Kubernetes and other container orchestration solutions?
How many employees work at your enterprise or organization? 2-100, n=62; 101-1,000, n=43; >1,000, n=67.

THE NEW STACK

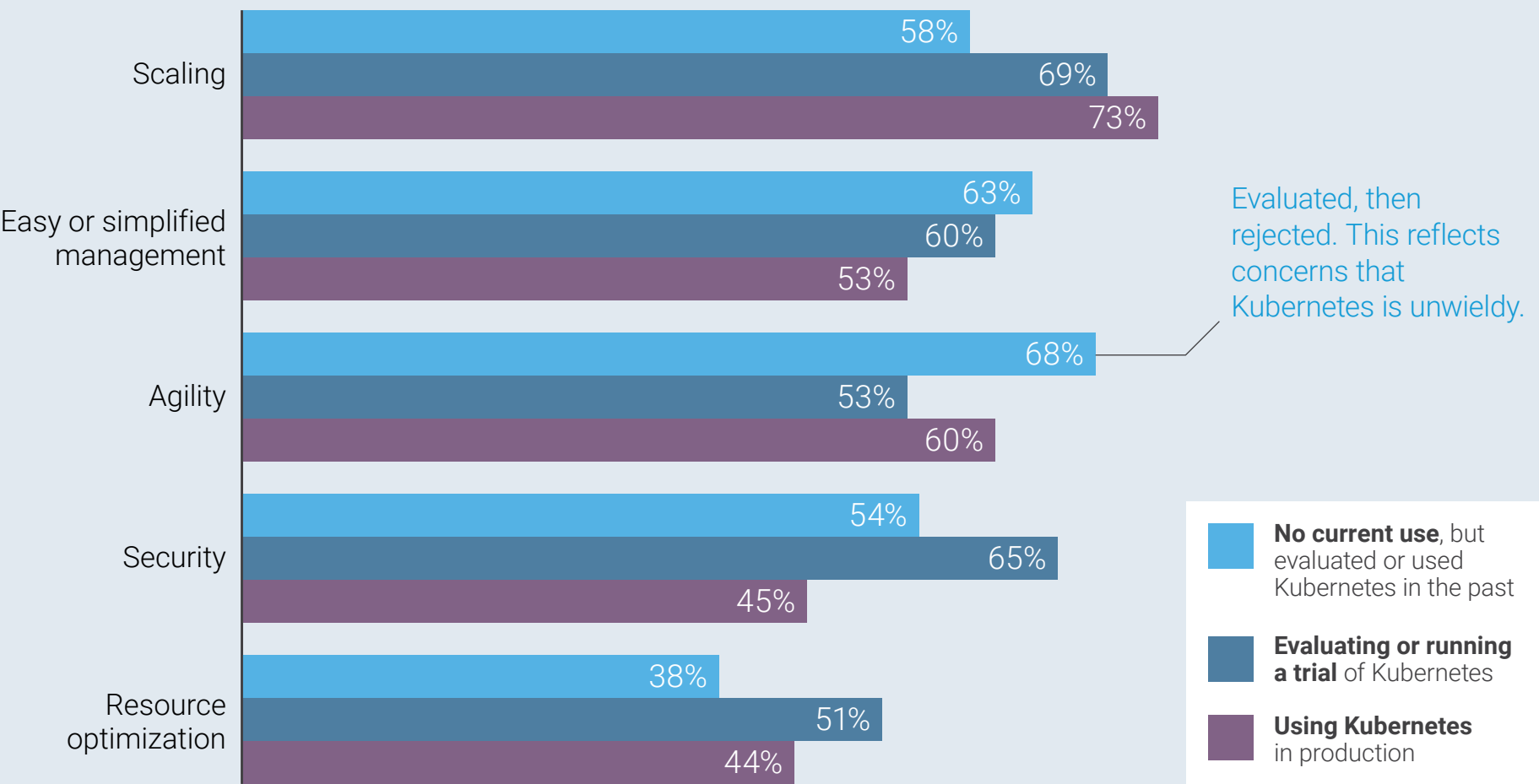
greater than 1,000 employees) gave higher priority to simple management, security and resource optimization than medium and small companies. In particular, easy or simplified management garnered 67 percent of respondents on the upper tier, compared with a combined average of 53 percent for the other two tiers. Respondents with middle-tier companies (100-1,000 employees) rated agility (74 percent) and scaling (74 percent) as equally essential.

Even though easy management was less likely to be an essential requirement for smaller companies, many respondents raved in their comments that Kubernetes was easy for them to use. They cited its APIs, its pluggable nature, and its ease of deployment across multiple clouds.

Next, we broke down responses to the decision criteria question by the current state of their Kubernetes deployments. Among those few respondents who evaluated and then rejected Kubernetes, 68 percent

Considered Essential When Evaluating Container Orchestration

(by Status of Kubernetes Evaluation)



Source: The New Stack 2017 Kubernetes User Experience Survey. Q. To what degree were the following considered when evaluating Kubernetes and other container orchestration solutions? No current use, but evaluated or used Kubernetes in the past, n~19; Evaluating or running a trial of Kubernetes, n~60, Using Kubernetes in production, n~197

THE NEW STACK

cited agility as an essential requirement. This may reflect their concerns that Kubernetes is a bit unwieldy, which is an opinion that we’ve heard expressed in IT communities and conferences as well. It’s still a young project, so perhaps it shouldn’t be surprising that folks do complain about documentation being incomplete, and that not everything always works right out of the starting gate.

Here are some critiques respondents shared with us about Kubernetes:

- “There’s a lot of rapid changes, which makes it hard to deploy a very large environment “across” as you have to continue going back to adjust things.” [101 - 1,000 employees]
- “Pain in the ass. Poor documentation on installing and operations. Components changing to fast to make sense of best options. Choices vary wildly across cloud providers. Difficult to choose based on limited info.” [>1,000 employees]

- “None of these are truly 100% production ready. K8s itself seems to have several issues running at scale and in production. Applications in containers/pods in K8s need to consider the environment and require tuning when porting over. Numerous K8s components are not proving as robust at scale during massive cluster events that can result in service outages in some cases. This arena needs further improvements and maturity, often too bleeding edge, resulting in the need to be constantly patching and upgrading to get features and fixes needed.” [>1,000 employees]
- “It’s a lot of hype with just enough cool stuff to make it not quite not worth the effort. An extraordinary amount of engineering has gone into the Kubernetes API with a healthy chunk of its functionality not exposed via the OSS command line tools and often poorly documented (e.g., RBAC). Perfect if your intent is to make money selling SaaS products. I’m keeping a candle lit that Kube doesn’t turn into another Orchestration-System-That-Shall-Not-Be-Named (*cough* OpenStack *cough*). :)” [unknown company size]
- “Lots of ways to do it wrong, no production level guides for setup. Easy tools leave much to be wanted. Automated tools were not good for production.” [2 - 100 employees]
- “Kubernetes is full of bugs and is not the application delivery Holy Grail everyone likes to think it is.” [101 - 1,000 employees]
- “Actually Kubernetes is hard to manage, I didn’t see that coming. After you step away from single master and a few minions, and your teams start to use the clusters you built, you will see what am I talking about.” [101 - 1,000 employees]
- “More headcount required to operate. Due to the rapid change in these platforms, more headcount is required to keep everything updated

and working effectively.” [101 - 1,000 employees]

Reasons to Use Kubernetes

Besides having general requirements, survey participants told us why they selected Kubernetes. Many cited its popularity among their key criteria. One commenter cast a negative light on that popularity, saying too many people “had drunk the Kubernetes Kool-Aid.” Some did cite the strength of its community, while others appreciated the strong level of support from recognized companies including Google and Red Hat.

One emerging theme was that Kubernetes was technically superior, at least presently. As one respondent explained:

- “It was the first container-centric orchestrator offering a complete set of features needed for building microservices style applications. Service discovery was leagues ahead of everyone else, and it came with configuration management features. The others seem to be just catching up now.” [2 - 100 employees]

Others appear to be expecting Kubernetes to maintain its current edge in the orchestration market on account of its perceived technical competitiveness. As others told us:

- “Kubernetes is awesome. Azure has a good grasp on where K8s is heading but their PVC implementation sucks and is behind. GKE is the strongest implementation out there. AWS is behind.” [101 - 1,000 employees]
- “Mesos does not provide an orchestration layer...besides that, the overhead involved to run Mesos is huge and the stability and scalability of the platform is not there. Cloud Foundry was dismissed due to it really being a “single-vendor” open source project with no clear mission and Pivotal seems to be moving towards K8s

themselves. Besides that, Cloud Foundry only enables hosting two-tier applications as their composition model is severely limited to using the app-broker bind model. None of the big-three cloud providers provide a good enough model captured inside a single hosted service. Google Cloud gets the closest. OpenShift is just abstracting K8s with limited added value, but at an additional cost. Hence using K8s open source, hosting it ourselves, automated the infra and K8s deployment to all platforms we need.” [100 - 1,000 employees]

- “Speed, scalability and security are important to us. We started with a homegrown solution based on earlier versions of Docker with Consul for service discovery over a bridged network. Things did not turn out well for us. The homegrown solution was hard to scale and maintenance was a nightmare. We played around with Apache Mesos, but it just did not suit our needs. When Docker Swarm was released we already had our cluster running K8s. Docker Swarm did not have the correct implementations for features we already had in K8s.” [101 - 1,000 employees]
- “Container orchestration is really hard to do right :). Most of the solutions are either too complicated (for example Mesos and DC/OS) or work incorrectly (Rancher, Kontena and other small players). Also there are “enterprise” solutions like Cloud Foundry which are unnecessary complex. We work in the IoT area and we have one more criteria for the container orchestration solution: it should be possible to move the system to bare metal. That’s why we use Nomad at the moment for IoT, but might evaluate K8s in the future.” [2 - 100 employees]
- “Not all platforms are equal in terms of capability, stability and scale. VMware misses the core services that other infrastructures provide. OpenStack != OpenStack in terms of compatibility of APIs between

versions and distributions. Google Cloud and AWS are the most stable; Azure is catching up. Bottom line is that there is a lot of diversity underneath, which imposes risk. K8s OSS does not test on all platforms, so we run validation of builds we want to adopt, building our own version comprised of the changes we need and passed validation. There is no vendor delivering a cross-platform! Heptio was that promise, which never happened, too bad.” [101 - 1,000 employees]

Roadblocks to Kubernetes Adoption

Kubernetes’ perceived edge in the container orchestration market, as young as that market is today, is neither definitive nor definite. Its survival may yet depend on competitors’ ability to match customers’ expectations for the essential requirements for orchestration. In the future, enterprises may look for solutions that are bundled or included with larger platforms, or they may simply accept those solutions once they’ve discovered they were already bundled with the platforms in which they’ve already invested.

The Kubernetes development community needs to address the inhibitors to its adoption, especially among evaluators who have yet to commit.

Inhibitors and Obstacles

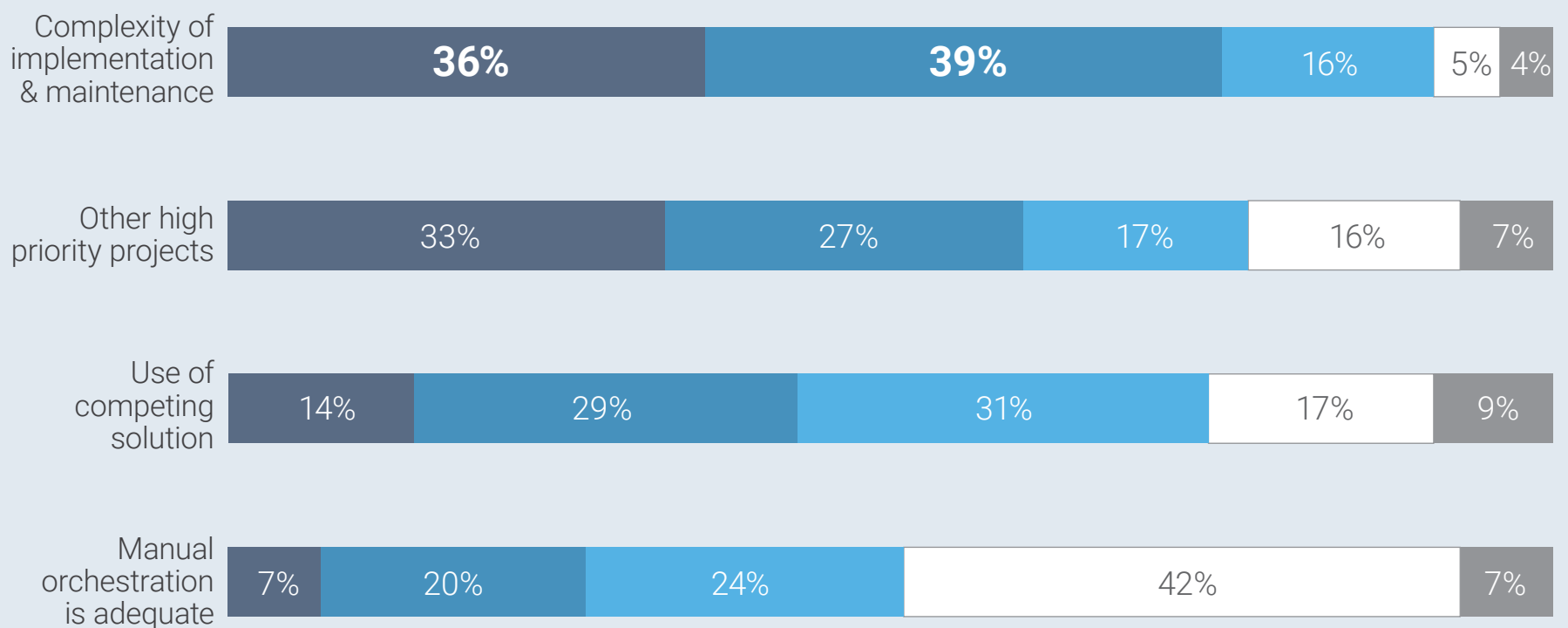
As we’ve seen, agility is more important to people who evaluated Kubernetes in the past, but chose another path. We explicitly asked this group, along with the group currently evaluating Kubernetes but not having yet committed to it, what is inhibiting their adoption?

We gave these groups a list of four categories of adoption inhibitors, and asked them to rate each one on a five-point scale, ranging from “To a great extent” to “Not applicable.” The complexity of Kubernetes implementation was inhibiting adoption to a great extent for 36 percent of

Inhibitions to Kubernetes Adoption

(Not Asked of Those Using Kubernetes in Production)

Reasons for Not Using Kubernetes



Impact of Reason on Decision



Source: The New Stack 2017 Kubernetes User Experience Survey.
Q. To what degree are the following inhibiting your enterprise or organization's future adoption of Kubernetes? n~133.

THE NEW STACK

respondents. Some three respondents of four in these subgroups cited complexity as an important inhibiting factor to some degree. Yet concerns about complexity were more pronounced among those whose organizations had evaluated and rejected Kubernetes.

The other big inhibitor was simply the higher priority for other projects. Some three respondents in five of these subgroups cited higher-priority projects elsewhere as an important concern, with one in three giving Kubernetes' low priority their highest rating. For these companies, Kubernetes adoption may have to wait until other organizations have found success with it.

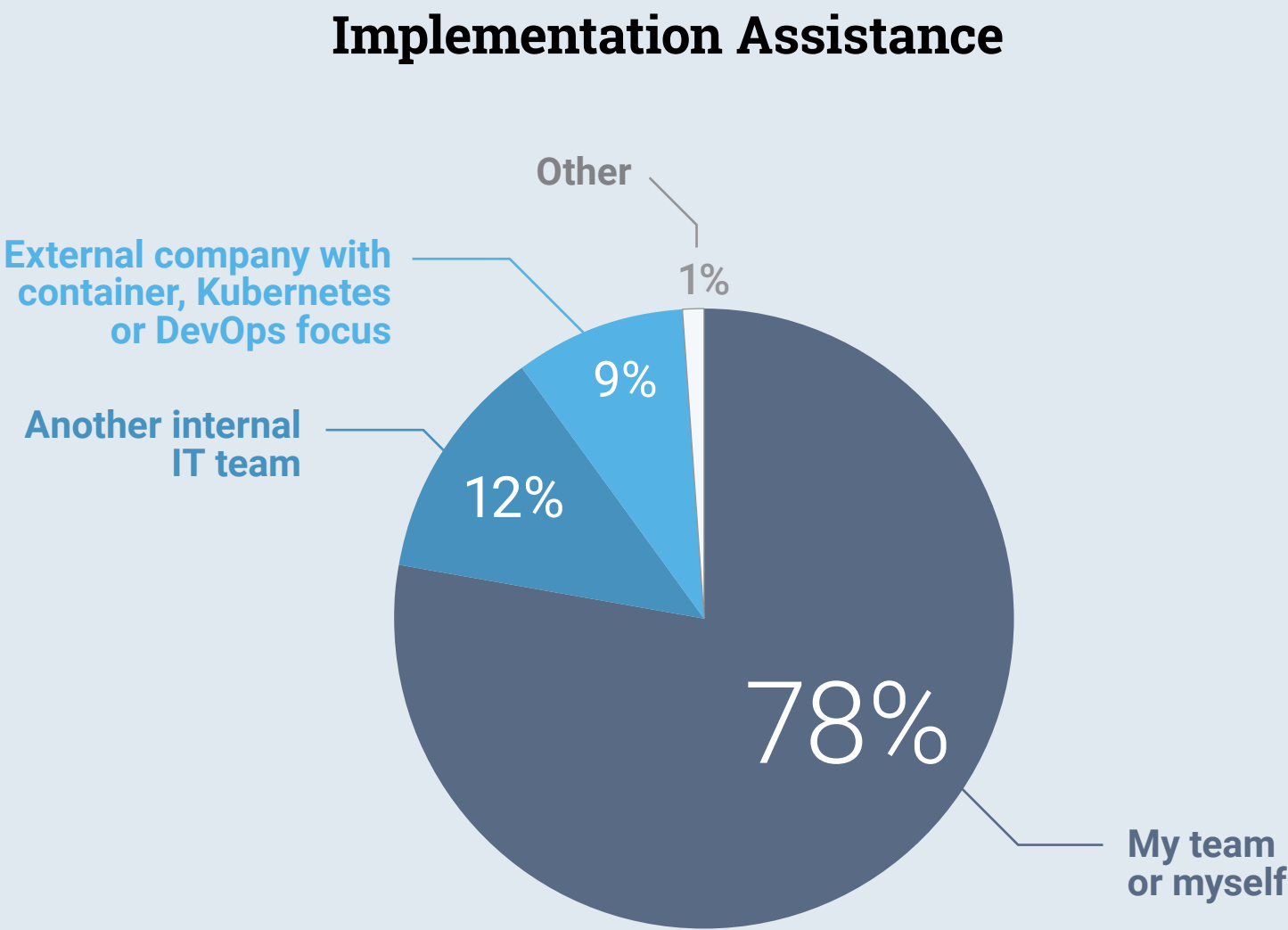
One other factor was much less likely to be holding adoption back. Respondents made clear their belief that manual orchestration is not an option, with only seven percent of respondents in these subgroups citing a preference for manual orchestration as their key inhibitor. Of those

whose organizations don't use Kubernetes, 42 percent said manual orchestration was no inhibitor at all. Only 14 percent cited their organization's use of a competitive solution as a significant inhibitor.

Implementation: The Good, the Bad and the Ugly

We've established that Kubernetes has required a reputation for complexity of implementation, particularly among institutions that have rejected deploying it in production. Is any part of that reputation shared by those organizations that have adopted it?

We asked some questions of adopters to figure this out. To give you a better idea of the extent of this group's hands-on relationship with implementation, some 78 percent of respondents told us they or their teams were directly responsible for implementing Kubernetes, as



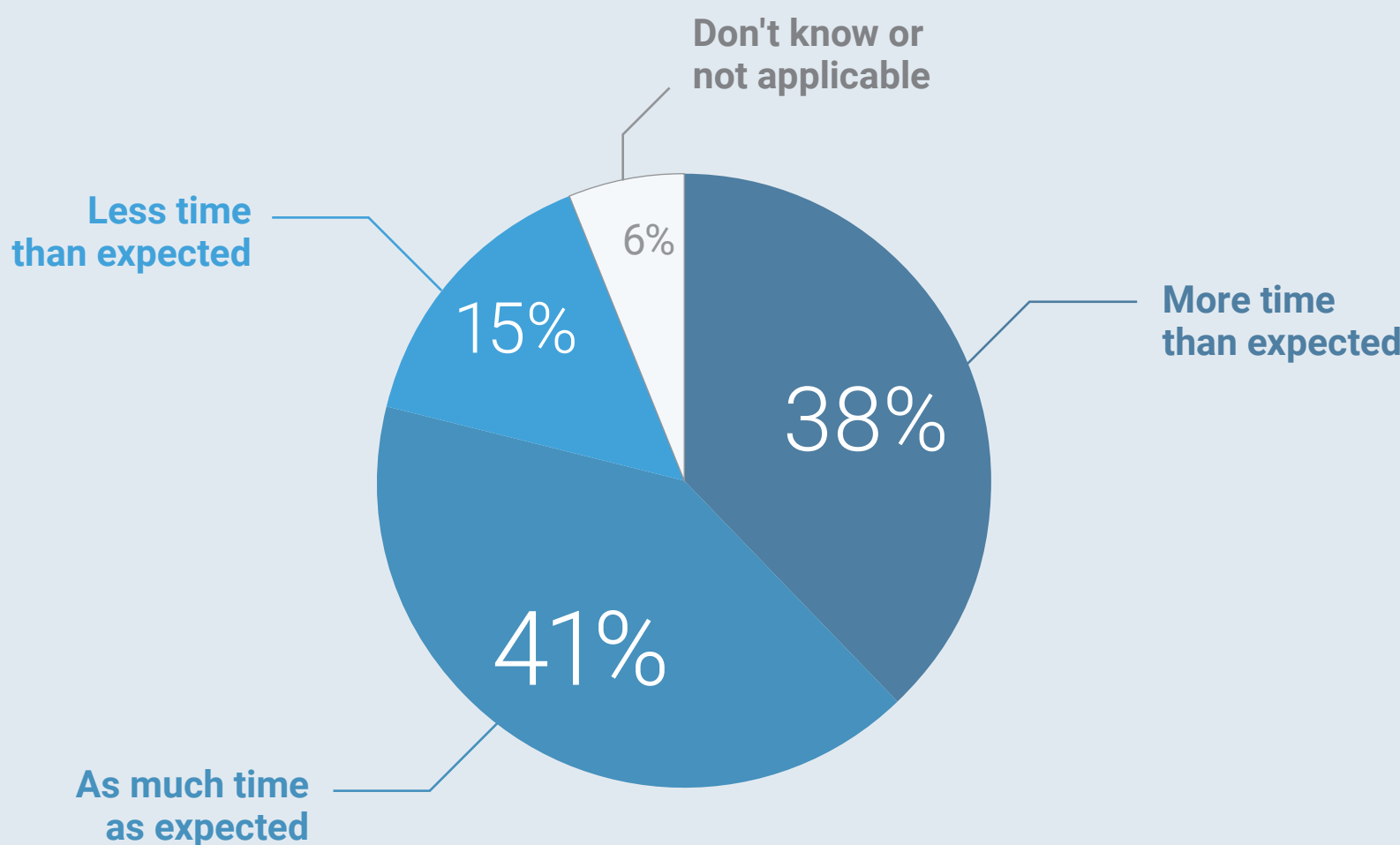
represented in the chart below.

Implementation Takes Longer than Expected

We asked these implementers whether their organizations' implementation of Kubernetes consumed as much time as they initially estimated, or instead more or less time. Some 38 percent of respondents said it took longer than expected, while only 15 percent said it took less time. Complexity may be to blame, if only partly.

For those who said Kubernetes implementation took more time than they expected, the percentage declined as their experience with implementation increased. Specifically, those with broad implementations registered 14 points lower than those just starting their implementation, for taking too much time. This clearly suggests that a good number of implementers — though certainly not all — do feel more comfortable with Kubernetes given time, once they've taken the plunge.

Meeting Overall Expectations for Implementation Time



Source: The New Stack 2017 Kubernetes User Experience Survey.
Q. Did the implementation take more or less time than expected? n=182.

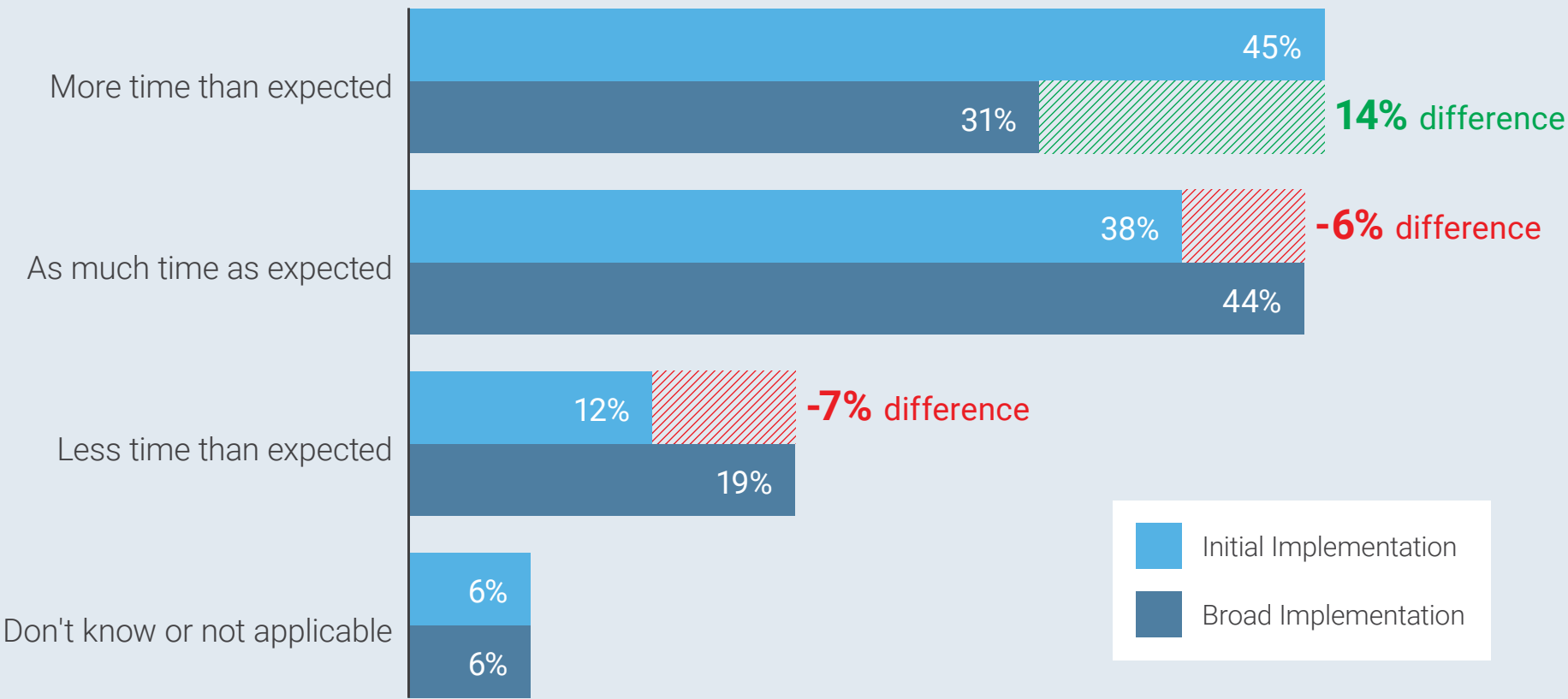
One commenter who works in a small business offered this bit of hopeful insight:

- “[It] took much more time than expected because of the complexity of deploying and managing the K8s cluster, and the poor documentation there was one year ago. Today is much easier.”

While the installation and configuration stages may be causing delays, several commenters explained that organizational dynamics were the biggest time drain. Here are some examples:

- “Organizational and procedural hurdles are hard to overcome. You need a company-wide initiative to even give this a chance.” [>1,000 employees]
- “We underestimated the amount of work it would take for everyone to adopt continuous delivery, but that had been overdue anyway.” [101

Meeting Expectations of Time Required:
Initial vs Broad Implementation



Source: TNS 2017 Kubernetes User Experience Survey. Q. Did the implementation take more or less time than expected? Initial Kubernetes Implementation, n=104; Broad Kubernetes Implementation; n=78. Due to rounding, percentages may not always appear to add up to 100%

- 1,000 employees]

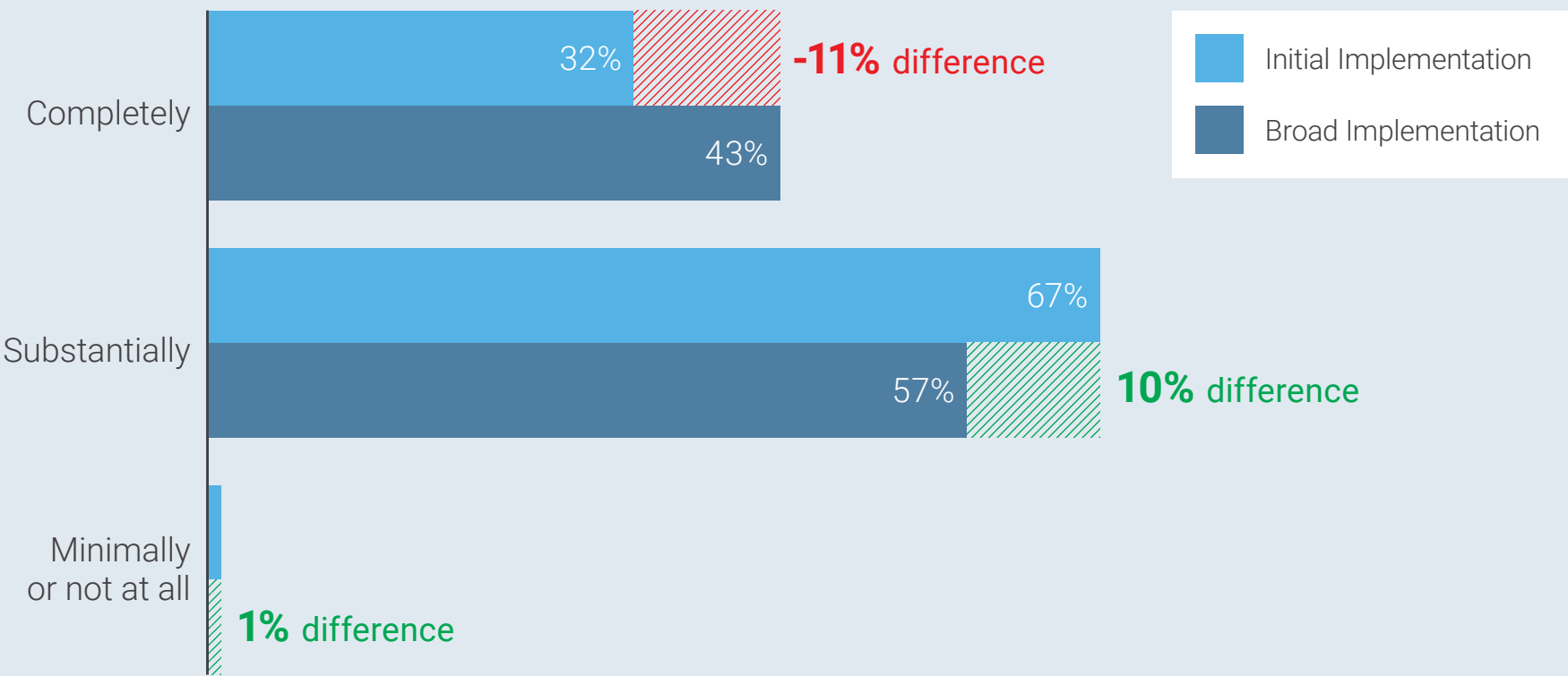
Obviously, the move to microservices architectures is a process that cannot easily be rushed.

Is There a Market Opportunity for Easing Implementation?

If Kubernetes is complex and takes time to implement, perhaps there’s an opportunity for emerging companies to offer a unique value proposition for helping out. Perhaps. But so far, not yet.

We asked participants in both the initial and broad stages of Kubernetes adoption whether the current status of that adoption has met their organizations’ expectations completely, substantially or minimally. While only 32 percent of initial-stage respondents say Kubernetes has completely met their companies’ expectations, some 43 percent of broad-stage respondents are completely satisfied. That’s certainly the

**Kubernetes Meeting Primary Goals:
Initial vs Broad Implementation**



Source: The New Stack 2017 Kubernetes User Experience Survey.
Q. To what degree has Kubernetes met the primary IT or business goals it was selected to address?
Initial Implementation, n=92; Broad Implementation, n=81.

right direction. And although fewer than half of the broad-stage group are completely satisfied, keep in mind that, at just two years of age, the Kubernetes ecosystem, or market, is a whole year younger than what some analyst firms consider a requirement for a mature market.

Some nine percent of total respondents told us they did rely on help and support from an external company to implement Kubernetes in their organizations. It appears that hiring such a company did have some impact, as only one in five of these respondents (20 percent) said the implementation took longer than expected — an improvement over the 38 percent of all Kubernetes users that complained about time consumed.

Only 31 percent of people with broad implementations said implementation took longer than expected — another improvement, this time over the 45 percent of those with initial implementations that complained about time consumed.

Overall, Kubernetes is winning the expectations game. According to 173 production-level users of Kubernetes, a full 99 percent said the platform has either completely or substantially met their IT or business goals. Granted, these were companies for which Kubernetes passed their evaluation phase. During an evaluation phase, the expectations for a product being evaluated typically become more closely aligned with that product.

While using a vendor distribution is supposed to make things easier (at least, so vendors say), just 47 percent of those who only cited a community-supported project, rather than a vendor-based product, said they were completely satisfied. With greater investment may come greater expectations.

Kubernetes Users Share Their Experiences

The consensus among participants in this survey is that Kubernetes reduces code deployment times, and increases the frequency of those deployments. However, in the short run, the implementation phase does consume more human resources. To address these issues, the structure of IT organizations may need to change.

Here are some comments participants shared with us about the impact of Kubernetes on their organizations.

- “300 deployments per day instead of... 0.5. Test cycle in the team gone from approximately 1 hour to 5 minutes. Enabler for continuous delivery.” [2 - 100 employees]
- “When evaluating DevOps roadmaps, K8s comes with so many features built-in that would require minimal effort and accelerate our roadmap as well as allow us to have a cloud agnostic solution. It also minimized cost and resources in use for the applications we were deploying.” [101 - 1,000 employees]
- “We reduced our infrastructure bill by four times. Now it just works and we don’t spend time debugging it. Developers can deploy their application easier and in a more sane way (using YAML descriptions)... We have ingress with SSL. We finally handle secrets properly. Big impacts on the security side.” [2 - 100 employees]
- “Loss of fear of introducing additional independent services, enabler of microservice architecture. Additionally, we can now spin up production-identical clusters for load and performance testing, and/or to validate large changes in infrastructure or architecture. A really powerful thing.” [>1,000 employees]

- “Kubernetes has enabled us to bypass our system engineers allocating resources for each new project and experiment. We’ve gone from spinning up trial runs in minutes from waiting days for VM resources.” [$>1,000$ employees]
- “It’s much easier for developers to get their applications running on the cluster now than before. Before, understanding a deployment system like Chef was necessary, which limited the number of people who could support and deploy applications. Now, most developers grasp the basic Kubernetes concepts and can deploy applications with relative ease.” [2 - 100 employees]
- “Devs have more ownership: Before to scale an app we needed to create a server, configure it, deploy the latest code and add it to load balancers. So it needed involvement of the infra guys (my team), the app guys, and it was too painful. Now they can just set the number of replicas and it’s done. Or even use HPA. Also, devs have the possibility to change configuration files (via configmaps) of web servers or monitoring daemon on their pod. Before, they needed to ask us, as those changes were handled via Chef. . . Ops team has more time: We are not a bottleneck for the operations just listed, and that gives us more time to do things and not maintain. . . We can easily reserve instances on AWS, since all fits in the K8s cluster. We couldn’t before, as when an app load changed, the servers in AWS needed to be changed and it was not easy. So we save money. . . The 90% of the pages we receive are capacity problems. With HPA and cluster auto-scaler, we can eliminate or reduce them significantly. For example: we only receive pages for things on our legacy infrastructure and not on Kubernetes, in general.” [101 - 1,000 employees]
- “We save \$800 month in CI/CD using Jenkins in K8s. We reduced 300% our deployment time process, and saving 15% of infrastructure cost

using autoscaling.” [2 - 100 employees]

Although success may be measured by Kubernetes’ overall impact on organizations and how they work, the initial point of impact comes at the time of setup. Here is just a sample of the keys to the platform’s success that participants shared with us.

- “Having kube-admin to take care of some initial setup really helped. Also, KubeCon’s conference videos and work by the Cloud Native Computing Foundation.” [>1,000 employees]
- “Having Google take care of the hardware, networking, logging, etc. I would have given up, if I had to do it all myself.” [2 - 100 employees]
- “The ability to create “operators” or replace part was key. For instance, we had to replace the kube-proxy with an iptables implementation before it existed in Kubernetes. Same thing for the Ceph RBD provisioner — I deployed an implementation in late 2014. Then, the quality of the API allowed for easy development of a pipeline manager optimized for our need (currently, less than 10 seconds between git push and updated deployment; heading for subsecond is the next challenge).” [2 - 100 employees]
- “Lots of communication across teams, many back-and-forth cycles of deploying applications and seeing what issues arise.” [101 - 1,000 employees]
- “Kubernetes is continually improving and bugs are getting fixed: It doesn’t feel like my issues are going into a black box and never heard. Also, K8s is written in Go, which makes writing patches very accessible.” [2 - 100 employees]
- “Linux knowledge, getting broad support from across the team, early application success.” [>1,000 employees]

- “[I recommend building] your own version from K8s upstream, leverage supporting tools and automate infrastructure and K8s deployment on the infrastructure to be able to rebuild, test, validate quickly.” [101 - 1,000 employees]

Other participants advised potential Kubernetes users to have more realistic expectations:

- “Developing in-house expertise in the underlying technologies has still been essential even though we’re using a heavily community-supported system. Application design and tuning for Kubernetes deployment presents its own set of challenges. Even though Kubernetes and its technologies reduce overall system complexity, there’s still a substantial amount of detail to understand when supporting and debugging production deployments.” [2 - 100 employees]
- “K8s is not a silver bullet and it still requires a lot of Ops. The multi-cloud vendor ideal is by far harder to achieve, even using a container orchestration environment like K8s.” [>1,000 employees]
- “Kubernetes is still very young in many places. It still has lots of sharp corners, for example cron jobs are pretty much useless for most normal workloads. But luckily it’s being reworked.” [101 - 1,000 employees]

One could easily make the argument that the Kubernetes ecosystem, such as it is, has yet to reach a reasonably mature state, and that any judgments we make at this point would certainly be preliminary at best. But it is remarkable that a major component of the world’s data center virtual infrastructure can evolve to this point, with so many satisfied implementers, just two years after the release of version 1.0.

RETHINKING THE DEVELOPER PIPELINE



In the chain of events that defines the modern evolutionary path of the application — a path that now includes microservices, persistent containers, orchestrators, monitoring tools and “kubelets” — when does the security part begin?

“There’s a lot more responsibility that’s on the developer, or at least in the developer’s workflow, to secure that application,” said John Morello, CTO for container security platform provider Twistlock. “In the new world of containers, your developers need to ... re-create the images that are vulnerable. And then they need to deploy those new images to replace whatever’s out there.”

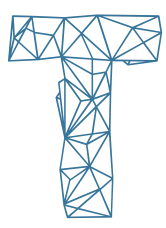
Learn more about Morello’s perspective on where security fits into the orchestrator-driven itinerary. [Listen to the Podcast.](#)



John Morello is the chief technology officer at Twistlock. As CTO, John leads the work with strategic customers and partners and drives the product roadmap. Prior to Twistlock, John was the chief information security officer of Albemarle, a Fortune 500 global chemical company, and spent 14 years at Microsoft.

BUYER'S CHECKLIST TO KUBERNETES

by **KRISHNAN SUBRAMANIAN, JANAKIRAM MSV, CRAIG MARTIN**



The rise of containerization, first championed by Docker, has changed the dynamics of deploying and managing software. Developers are using containers on their local machines, configuring a variety of environments for their own purposes without any reliance whatsoever upon IT. Meanwhile, enterprise IT teams are under pressure to support all levels of application deployment, from developers' laptops to their own production environments.

Containerization fulfills one key goal of modern IT: It shifts the objective of deployment from virtual machines (VMs) and virtual servers to applications and workloads. Developers can focus on building applications and operators can switch to managing applications rather than managing VMs, in order that the applications contained within them run smoothly.

It's a worthy goal. But making this mindset shift requires organizations to absorb and integrate a concept they probably would have never considered before, at least not voluntarily: orchestrated workload distribution.

We've compiled several lists of the most important considerations that various stakeholders should make — cultural, organizational, practical, and technical — before you make an affirmative decision to deploy Kubernetes. Contributing their expertise to this chapter are:

- [Krishnan Subramanian](#), principal analyst, Rishidot Research, LLC.
- [Janakiram MSV](#), principal analyst, Janakiram & Associates.
- [Craig Martin](#), senior vice president of engineering, Kenzan.

Considerations Before Selecting Kubernetes

Kubernetes is but one permutation of orchestrated distribution. At the moment, it's the one with the largest plurality of users, but this horse race has just started. Docker Swarm and Mesosphere Enterprise DC/OS are competitive players. If you do select Kubernetes, it will be after you've made a multitude of important considerations.

What the CIO Needs to Consider

The typical goals of the IT department are to provide the organization with robust and reliable applications. Yet as IT faces the challenges of modernization for today's economy, the chief information officer of the organization not only expects IT to provide robustness and reliability, but also become more agile and more efficient with respect to resource consumption. Kubernetes helps IT in its quest to optimize resources and operate at a much higher scale than any time in the past.

The key to the success of the modern enterprise lies with two key areas of empowerment:

1. How IT empowers developers by delivering the foundational services they need, at the scale the business needs.

2. How IT empowers DevOps with the tools and support they need to deliver software to customers with greater agility.

Kubernetes is the secret sauce transforming IT from the gatekeepers to the innovators of the modern enterprise. Here's what the CIO will need to consider with respect to how Kubernetes can serve as the vehicle for that transformation:

Business Considerations

- **Value assessment:** It's important that the CIO make a strategic assessment of the business value of IT transformation, and how containers and Kubernetes can impact the organization's transformation. An organization may find business value from something that adds revenue to the organization, or something that gives them strategic competitive advantage. Kubernetes is not a magic pill that can cure all IT woes. CIOs should get a buy-in from all stakeholders, which requires a careful discussion of potential disruptions, and a plan for how to mitigate them as Kubernetes is being rolled out. Stakeholders here may include IT administrators, developers and business users. Modern IT involves a buy-in from all these stakeholders. This helps usher in cultural change, along with the technology and architectural changes brought in by modernization.
- **Legacy assessment:** Kubernetes supports legacy applications through stateful storage options, although these are typically not ideal workloads for Kubernetes. CIOs should prioritize the right high-value workloads for migrating to Kubernetes, from both a technical and business perspectives. There may be a cost associated with architectural changes to applications, and CIOs should consider these costs as well. For instance, a mission-critical application might get disrupted — leading to a loss in business value — if it gets moved to Kubernetes just for the sake of using Kubernetes.

- **Process assessment:** Using a platform like Kubernetes can provide agility to IT and can help deliver business value fast. The CIO should think through their organization's entire value delivery process, taking into account potential pitfalls, and deciding if the investment in Kubernetes is the right one. ROI can be maximized when architectural changes to applications can be coordinated along with the move to containers and Kubernetes.

Technology Considerations

- **Paradigm shift:** Using containers and orchestrators requires a mindset change among IT decision makers, especially CIOs. Instead of thinking about the reliability of applications, the IT professional needs to think in terms of their resilience. It is important for CIOs to usher in this mindset change for both IT and developers.
- **Architecture shift:** When infrastructure and application components are treated as cattle instead of pets, you'll soon need to rethink existing application architectures. CIOs should be prepared for this shift, and get buy-in from developers well ahead of time.
- **Placement shift.** Kubernetes may be deployed across multiple cloud providers, or with on-premises infrastructure. The CIO should develop a deployment strategy based on the organization's needs first, and the needs of the infrastructure later.
- **Storage shift:** It is important for the CIO to identify whether the organization's applications require stateful storage. They should ensure that the needs of stateful, storage-oriented apps — needs that won't disappear under Kubernetes — will be supported.
- **Declaration shift:** With Kubernetes, you always need to consider the system as a whole, and tap into its declarative model for deployments. This requires a mindset change both at the infrastructure and

application levels. Kubernetes espouses a declarative model, as though it were asking you, “tell me what you want, and I’ll try to do it,” as opposed to the classic, imperative model, “please instruct me as to how I should create exactly what you want.”

People and Process Considerations

- **Talent considerations:** A move to Kubernetes will require cross-functional talent. The CIO should have a strategy in place for either hiring new talent or retraining existing talent, not only to better comprehend the new technologies, but to embrace the process changes that come with a move of this scale.
- **Cultural considerations:** Just embracing Kubernetes is only one leg of the bigger journey. The time for complete compartmentalization of development teams from operations teams in the modern enterprise, is over. And although there are a number of variants of this concept called DevOps, they all have in common the notion that they should share responsibility, and communicate more directly, with each other. Kubernetes is not a collaboration platform for Dev and Ops, or for Dev and Sec and Ops, or whatever syllabic mixture is in vogue today. But adopting it properly, and embracing the concept of distributed systems orchestration, does require the people who create applications and the people who manage them to, more than infrequently, have lunch together and swap stories. It is the responsibility of the CIO to empower stakeholders to facilitate this communication, so that these teams don’t have to wade through bureaucracy just to have a constructive chat.
- **Failure considerations:** Containers and orchestrators provide an opportunity for developers to experiment and fail fast. The CIO should mandate this allowance, and empower stakeholders to experiment in such a way that failure becomes the first step toward remedy and improvement.

What the IT Implementer Needs to Consider

Kubernetes is very powerful at container orchestration, but it isn't necessarily a perfect fit for every development context. Key stakeholders within organizations should ask themselves these questions first:

Will your applications need a distributed architecture (e.g., for microservices)?

While Kubernetes can work in a monolithic infrastructure, its focus is on orchestrating a large number of small services at big scale. What you will need to consider is whether the services you run today, plus those you plan to run in the future, can be decoupled from the application that uses them. Put another way, can the code that does the work be deployed behind an API? If so, you can use Kubernetes to orchestrate those services separately from the clients that call upon them. That separation is essential to making those services scalable.

Are monitoring tools in place to support a Kubernetes deployment?

Infrastructure monitoring helps ensure the health and availability of resources — storage, CPU, network — for the applications it serves. If you don't have such tools in place already, you should invest in a robust monitoring mechanism that can track the health of underlying nodes, as well as that of the workloads running in Kubernetes. There are open source and commercial monitoring tools that integrate well with Kubernetes environments. Start considering monitoring tools, along with your other tools, right away. We touch more on monitoring considerations in the next chapter, as Kubernetes can bring about unique challenges.

Are your applications container ready?

Containers are different from virtual machines. Everyone in the organization — including developers, system administrators and DevOps

practitioners — should have a basic understanding of containers. If they don't yet appreciate the business value delivered through containerization (and not everyone will at first), they should at least respect the leaders of the organization and their reasoning behind this investment. Teams should first adopt containers in non-production environments such as development, testing, QA and staging.

Are your people container ready?

Adopting Kubernetes comes much later in the transition process than adopting containers. Docker adds value to dev/test environments and to continuous integration / continuous development (CI/CD) processes. That value should already have been added before starting with Kubernetes — or, for that matter, any orchestrator. A full appreciation and acknowledgment of the business and technical value of containers are prerequisites before you can use Kubernetes effectively to manage containerized workloads in production. Management should be on board with the benefits of adopting container technologies. Most importantly, all stakeholders should be trained on working in a distributed systems environment with containers. Google offers certified training for Kubernetes professionals, specifically for Google Cloud Platform.

Are you planning to migrate legacy applications into Kubernetes?

The migration approach you choose for legacy migrations might be challenging, whatever it may be. One common approach, for example, would be to deploy an API Gateway, then decompose the monolith into Kubernetes pods one feature set at a time, over an extended period of time. It's an effective and manageable approach, but it's not necessarily easy.

Are you planning, instead, to start with a fresh greenfield deployment?

The challenges of developing and deploying applications to a completely

fresh Kubernetes environment are altogether different. You may be freer to take risks with what are essentially completely new technologies. At the same time, you'll be encountering the pain points along with everybody else.

Will you choose a commercial version or a community distribution?

Kubernetes is available as a stock open source distribution, or as a managed commercial offering. Depending on your internal IT team's skill set, you may either choose the stock open source version available on GitHub, or purchase a commercial orchestrator based on Kubernetes from a vendor, such as [CoreOS](#) and Canonical, that offers professional services and support.

Are you ready to invest time and energy in building your own container images?

A container is based on a pre-configured image. Such an image typically includes the base operating system. Unless the contained application is a compiled binary, the image may also include the libraries and other dependencies upon which it may rely. Your organization may wish to invest in a private registry, which stores both base images and custom images. Some commercial registries come with image scanning and security features that scan for vulnerabilities. Even so, a [recent study](#) examining images stored on the Docker Hub registry found some four images in five contain at least one documented security vulnerability. So you may choose instead to completely compile your image components from binary files, using libraries your organization knows and trusts to be safe. Alternately, you might consider an architectural approach suggested by CoreOS engineer [Brian Harrington](#), called [minimal containers](#) — a more spartan approach to assembling containers.

Is your storage ready for high-performance workloads?

A Kubernetes cluster may be based on distributed file systems like NFS,

Ceph and Gluster. These file systems may be configured on solid-state storage backends that deliver high throughput. Stateful applications running in Kubernetes can take advantage of these underlying storage primitives, which can make all the difference for running production workloads.

What is your expected level of uptime?

Your customers’ service level agreement (SLA) requirements will have a major impact on every aspect of your Kubernetes deployment: how you configure your environment, how you configure each application, how much complexity you can withstand, how many simultaneous deployment pipelines you can support ... and the list doesn’t stop there. All of these variables impact the total cost of your application. For a bit of context, Table 4.1 below explains the expected downtime for each “9” in your availability goal (from [a book by Susan J. Fowler](#)). It’s up to you to determine the amount of effort required for your organization to achieve each level.

Do you have, or plan to have, a release management system?

One of the key benefits of moving to Kubernetes is automating the deployment of applications. Deployments in Kubernetes support rolling updates, patching, canary deploys and A/B testing. To utilize these

“Nines of Availability” Downtime Allowance				
Availability	Per Year	Per Month	Per Week	Per Day
99.9%	8.76 hours	43.8 vminutes	10.1 minutes	1.44 minutes
99.99%	52.56 minutes	4.38 minutes	1.01 minutes	8.64 seconds
99.999%	5.26 minutes	25.9 seconds	6.05 seconds	864.3 milliseconds

TABLE 4.1: How downtime in “nines” translates into real time.

capabilities, you should have in place a well-configured build automation and release management platform. Jenkins is one example of a broadly deployed automation tool that integrates well with Kubernetes, by building container images from source code that can be pushed to both production and nonproduction environments.

Are you prepared for all the logs?

Kubernetes supports cluster-based logging, allowing workloads to log container activity in a centralized logging destination. After a cluster is created, a logging agent such as Fluentd can absorb events from the standard output and standard error output channels for each container. Logs generated by such providers may be ingested into Elasticsearch and analyzed with Kibana.

What Kubernetes Needs for Production

It's no longer always the case that a production environment must be physically, or even virtually, barricaded from a development environment. That said, Kubernetes will have requirements for production above and beyond what's needed in a development or a testing environment. Here are the key factors you should evaluate, and how they differ.

High Availability and Resilience

The meaning of the phrase "high availability" may differ depending on its context. So for this discussion, we'll focus on deploying Kubernetes in such a way that it can survive a partial subsystem failure. This quality of survivability is often referred to as resilience (and sometimes as resiliency). Remember, Kubernetes is not the whole container system, but an entity deployed in a data center. It's subject to the same operating conditions and perennial threats as any other data center software. When we think about high availability in a Kubernetes environment, we tend to divide it into three distinct aspects:

1. **Infrastructure availability** determines whether the base infrastructure that Kubernetes runs on is highly available and distributed regardless of the environment.
2. **Kubernetes availability** ensures that the environment never has a single point of failure, and that the components of the orchestration system are properly and proportionally distributed along the same lines as the infrastructure.
3. **Application availability** focuses on the application and ensuring that all pods and containers are achieving the correct level of availability. Based on the needs of the application, the application components will be loosely distributed across the Kubernetes environment, which itself is distributed across the infrastructure.

Since Kubernetes may be installed on several public and private Infrastructure as a Service (IaaS) platforms, the capabilities and properties of those platforms collectively frame a Kubernetes installation's inherent resilience. For example, what is the level of partitioning an IaaS may provide?

Your public cloud's IaaS platform is made up of subdivided deployment locations. AWS calls these locations "regions" and the subdivisions "availability zones," and other major cloud providers have followed suit. Kubernetes may be deployed across those zones. But by default, Kubernetes always deploys a single master, regardless of how many zones you have. In a high availability scenario (which is the point of multiple zones anyway), having the master and the control plane on a single zone creates a single point of failure.

Several vendors, and other participants in the Kubernetes community, have come up with high availability mode installation processes, where there are multiple masters in the cluster, and preferably one master per

zone. This way, any container runtime at any location can survive one of those zones failing. So if your IaaS has two zones, you should have at least one master per region for high availability.

The most common, and typically most recommended, approach for handling this is with a multi-master Kubernetes environment. In this scenario, only one of the masters is “elected” as the live master, so if it goes down, a new master is elected. This new master is then used for handling all scheduling and management of the Kubernetes environment.

Kubernetes clusters are essentially sets of associated worker nodes. Since the generator of your business value is running on these clusters, it makes sense that each cluster is deployed across as many logical partitions as are available.

NOTE: It's considered best practice to dedicate your master node to managing your Kubernetes environment, and use worker nodes exclusively to host and run all the pods. This allows for a full separation of concerns between Kubernetes and your applications.

With a private IaaS, you can still partition servers into zones. While each private IaaS configuration may be as unique as a snowflake, there are some concepts common to all of them. Whether your IaaS is public or private, your storage and networking partitioning should correspond to your compute partitioning. In the data center, you would have redundant storage area network (SAN) capabilities and redundant top of rack and end of rack networking capabilities that mirror the redundancy in your blade enclosures. This way, you can provision redundant network, power, compute and storage partitions. With a public IaaS, each service provider handles storage and networking capabilities differently, though most build in partition redundancy. As for networking, you should make use of your IaaS' load balancer for exposing Kubernetes service endpoints.

Ultimately, all this low-level resilience from your IaaS allows Kubernetes to support replica sets, which ensure that specified numbers of pod replicas are consistently deployed across partitions. This way, Kubernetes can automatically handle a partition failure. Recent editions of Kubernetes have implemented a kind of super-construct for replica sets called deployments, each of which involves instantiating what's called a **Deployment** object. This object instructs a “deployment controller” to manage replica sets automatically, refreshing them at regular intervals so that they are as close as possible to the desired configuration that you declare.

This way, your pods maybe rebalanced across the surviving zones in an outage event, without an operator having to perform special actions. To be more direct about it, you can specify the **failure-domain** for a node as an explicit annotation, when you declare the node. You can tailor the scheduler to your specific needs, and ensure that it will replicate across nodes or across your specific infrastructure.

One of the ways that we have accomplished this in the past is through **nodeAffinity** (formally **nodeSelector**, [changed in Kubernetes 1.6](#)), which tells the scheduler exactly which nodes should be impacted when a scheduled event occurs. This ensures that application-level availability aligns with the infrastructure partitioning, thus removing or at least reducing downtime should the infrastructure fail.

Networking

The Kubernetes network model hinges on one important fact: Every pod gets its own IP address. This address is part of a network inside the Kubernetes cluster that allows all pods to communicate with each other. Because Kubernetes supports a modular implementation of software-defined networking (SDN), you have several choices available to you.

While it is indeed possible for you to implement a Kubernetes cluster without the use of an overlay network, its absence can impose potentially significant limitations and configurational overhead. In such a scenario, the infrastructure would need to be aware of all the cluster IP addresses, and be able to route to all of them. For an IaaS such as AWS, this can be quite limiting.

An overlay network gives you an easier means for managing IP routing inside Kubernetes clusters. This virtual construct separates the internal cluster network from the infrastructure on which Kubernetes is running, usually with some form of encapsulation. For simplicity, the most commonly implemented network overlay is CoreOS' [Flannel](#). There are others, such as [Calico](#) and [Weave Net](#), which implement not only IP Address Management (IPAM), but also network policies for controlling traffic.

Your selection of SDN has consequences for both cluster size and overall performance, and may depend upon the infrastructure provider on which the cluster is running. Each network overlay implementation has different modules for different infrastructure types and cluster configurations. The specific needs of each application will differ from one another, but here's the basics of what you need to know.

- [Flannel](#) is very flexible as it supports many different protocols and networking models. Marshaled by CoreOS, Flannel is also the most secure in our experience, and is the easiest to support distributed data needs.
- [Project Calico](#) is the most well-performing of the three options. We have consistently seen that Calico can outperform the others in raw speed. But what you gain in performance, you lose in features, as Calico doesn't support as many security features, protocols or subnet restrictions.

- [Canal](#) is an open source hybrid option launched by the Project Calico team, and now shepherded by Kubernetes migration platform provider [Tigera](#). It integrates the network policy management functions of Calico with the network connectivity components of Flannel.
- [Weave Net](#) is similar to Flannel with some small differences in critical features. It supports [NaCl](#) (pronounced “salt”) for encryption, it has better subnet restrictions, and supports partially connected networks (e.g., crossing firewalls). Those gains come at a slight loss in a few areas, as it tends to be less robust in distributed data needs or even in supporting transport layer security (TLS).

Storage

Containers are best suited for 12-factor stateless applications, where services respond to requests, then blink out of existence leaving nothing behind. This is, as you can imagine, a very new — and for many enterprises, foreign — way of presenting applications. And it simply cannot translate to the storage-heavy models of existing applications whose life cycles have yet to be fully amortized.

Kubernetes supports stateful and stateless applications simultaneously, by adding support for persistent volumes — specifically, the **PersistentVolume** subsystem — along with additional support for dynamic provisioning. This allows support for legacy and other stateful applications on Kubernetes clusters, thereby making Kubernetes an attractive candidate for enterprises. The orchestrator offers support for volumes that are attached to pods, as well as external persistent volumes.

A volume in Kubernetes is different from the volume concept in Docker. While a Docker volume is attached to the container, a Kubernetes volume is related to a pod. So even if a container inside the pod goes down, the

volume stays on. However, a Kubernetes volume is still ephemeral, meaning that it will be terminated when the pod is terminated. In embracing Kubernetes, you have to keep these two equivalently-named concepts distinct in your mind.

Kubernetes offers support for stateful applications by abstracting away storage, and giving an API for users to consume and administrators to manage storage. A volume is externalized in Kubernetes by means of the **PersistentVolume** subsystem, and **PersistentVolumeClaim** is how pods consume **PersistentVolume** resources in a seamless manner. Specifically, it creates a kind of tie between the containers inside pods, and a class of volume that will survive beyond the life cycles of those pods.

Kubernetes offers logical file system mounting options for various storage offerings from cloud vendors — for instance, Network File System (NFS), GlusterFS, Ceph, Azure SMB and Quobytes. The Storage Special Interest Group (SIG) is currently working on ways to make adding support for new storage services and systems easier, by externalizing the plugin support.

You should make sure the storage needs of your existing applications are supported before you begin the process of transitioning them into a distributed systems environment like Kubernetes.

Security

Containerization changes the entire concept of security in the data center. Orchestration changes it yet again. So it's fair to say that you are probably already contending with, and perhaps managing, your data center's transition to a new security model. The insertion of Kubernetes in this picture affects what that model may be changing to.

Security topics pertaining to workloads running in the context of a Kubernetes environment fall into three categories:

- **Application level:** session modeling, data encryption, throttling.
- **Platform level:** key management, data storage, access restrictions, distributed denial of service (DDoS) protection.
- **Environment security:** Kubernetes Access, Node Access, Master Node Configuration, encrypted key/value storage in etcd.

Node Access

Quite simply put, very few people should have direct access to a Kubernetes node. Every time you grant such access to someone, you are incurring risk. Instead of accessing the host directly, users can run `kubect1 exec` to get visibility into containers and their environments without introducing a security vulnerability.

If you need fine-grained, container-level security, operators can fine-tune access via Kubernetes' [authorization plugins](#). These plugins enable restricting specific users' access to certain APIs and preventing accidental changes to system properties, such as scheduler options.

Namespaces

Kubernetes allows for applications to be installed in different namespaces. This creates a lightweight boundary between applications, and helps to appropriately compartmentalize application teams. Such a boundary serves to prevent accidental deployment to the wrong pod or cluster, and enables the establishment of firm resource constraints (quotas) on each namespace. For example, you can give each app composed of microservices its own namespace in production. This allows for a single Kubernetes environment which hosts many applications, without the risk of collision between the applications (via namespaces).

A namespace acts as a logical boundary for workloads — it limits the breadth of an application to just that part of the system to which the same

names apply. More technically speaking, namespaces represent multiple virtual clusters backed by the same physical cluster. This makes containerization possible in the first place.

Resource Quotas

A [resource quota](#) provides constraints that limit aggregate resource consumption per namespace. Your enterprise IT team should create multiple namespaces, each with its own quota policy. This way, each policy may restrict the amounts of CPU, memory, and storage resources that a workload may consume.

In addition, a resource quota can protect against a single pod scaling to the point where it eats up all of your resources. Without a resource quota, it would be pretty easy for a malicious source to denial-of-service (DoS) attack the application. At the highest level, a resource quota may be set at the pod, namespace or cluster level to monitor CPU, memory, storage space or requests. Your specific needs will be determined by your applications and available resources, though it is imperative that you establish resource quotas.

Network Segmentation

Running different enterprise applications on the same Kubernetes cluster creates a risk of one compromised application “attacking” a neighboring application — not by hacking, but instead by passively interfering with the flow of traffic in the network they share. Network segmentation ensures that containers may communicate only with other containers explicitly permitted by policy. By creating subnets with firewall rules, administrators can achieve the right level of isolation for each workload running in a Kubernetes cluster.

Segmenting networks ensures that no two systems can share resources or call the wrong pods, which can be a very powerful way to secure

applications. But be careful: it's very easy to fall victim to temptation and overdo your segmentation policies, leading to an environment that's not only more difficult to manage, but produces a false sense of security among the development team.

Secrets Management

Kubernetes has built-in storage capability for secrets — discrete values that are typically pertinent to an application as it's running, but should not be shared with other applications, or with anyone else. When you define something as a secret, it is stored independently from the pod, ensuring that it is encrypted at rest. A pod is only granted access to secrets defined in its pod definition file. On container startup, secrets are provided to the container so they may be used appropriately.

Kubernetes provides a mechanism for making secrets available directly to a container by way of an environment variable. Specifically, you define the [`secretKeyRef`](#) variable in your pod definition file.

As a supplement to Kubernetes secrets, you may choose to use another secret management tool, such as Vault. An application dedicated to key management typically offers more robust features, such as key rotation.

Kenzan's View: “At Kenzan, we're very hesitant to use the `secretKeyRef` approach, as it's not too many steps removed from writing secrets directly into a properties file whose name and location everyone knows. Arguably, secrets between applications aren't necessarily secrets between people, or about people; but it's never a good idea to offer directions and a map to something intended not to be seen. Maybe there are use cases where this makes sense, but we would advise using `secretKeyRef` with caution.”

Access Management

With the exception of network segmentation, there is little security

between two pods. For many data centers this is a serious issue, because a bad actor could end up accessing the cluster, or at the very least triggering a networking error, and then all services would be exposed.

You may find it best to enact some level of internal security control to validate that pod A can call pod B, assuming these interactions are permitted by policy. Something like a JSON Web Token (JWT) claim (a JSON-based assertion used to help validate authenticity) with a short-lived, signed key rotation tends to work nicely here. This allows security needs to be very granular for each specific endpoint by providing roles within the JWT, and also ensures frequent JWT private key rotation (we recommend every minute or two). Using this model, anyone who did manage to penetrate through defenses and gain access to the network would still be unable to place a call successfully without the signed JWT. And a signed JWT is only valid for a minute or two.

In Conclusion

You've just seen a number of the key factors that pertain to the everyday care and maintenance of Kubernetes, as well as the network with which it is designed to interoperate. You and your organization would probably rather not make a serious commitment to a deployment decision that affects all aspects of application staging, monitoring and operations without having had more than a glimpse of what to prepare for. The next chapter will give you a preview of coming attractions, if you will: a snapshot of the operation of the orchestrator in a real-world production environment.

CLOUD-NATIVE APPS LEAD TO ENTERPRISE INTEGRATION



How companies are viewing Kubernetes is starting to change, as often happens when people start using the platform. Brian Gracely of Red Hat OpenShift maintains it is those initial applications a company develops that are leading to an enterprise shift to adopt Kubernetes on a larger scale.

Often, companies will start with a mobile app as the way to get started. This application development process helps teams understand what is necessary to really create a great experience for the user. The cloud-native app serves as a way to integrate with existing enterprise environments. Now the team is moving beyond a single app to a more holistic view that changes the way a company goes to market or even interacts with customers.


Learn more about what Gracely and the OpenShift team have discovered about integrating the modern software development mindset with the enterprise. [Listen to the Podcast.](#)



Brian Gracely is director of product strategy at Red Hat, focused on OpenShift. He brings 20 years of experience in strategy, product management and systems engineering. Brian co-hosts the award-winning podcast, [The Cloudcast](#), and the Kubernetes-focused podcast, [PodCTL](#).

ISSUES AND CHALLENGES WITH USING KUBERNETES IN PRODUCTION

by **CRAIG MARTIN**

 In this chapter, we will address issues surrounding the deployment of a Kubernetes environment in production. It's a big deal. It takes a lot of focused, deliberative consideration. There will be trial and error involved, which means you will need to be prepared for the error part. That's where resilience comes into play. Luckily, container orchestration was invented with resilience in mind.

To help guide you through the process of picturing Kubernetes at work in your own organization in production (not just in development), we've enlisted the help of Craig Martin, senior vice president of engineering at software engineering services provider [Kenzan](#). Martin and his team have helped enterprises set up continuous delivery processes, often involving Jenkins, and more often in recent years Kubernetes.

The Kenzan team will share with you the everyday facts of life and work they urge their own clients to consider before embarking on a transitional course that brings Kubernetes into their production environments. Here's some of what we'll cover:

- The tools and methods used for monitoring the Kubernetes platform and the applications it runs at scale.
- The methodologies required for logging system events as they happen.
- The tools and methods you need to put Kubernetes automation to work.
- The resources that Kubernetes and applications need from each other.

Monitoring Kubernetes at Scale in Production

The concept of monitoring a distributed systems environment is completely different from monitoring a client/server network, for a reason that becomes retroactively obvious once it's discovered: The thing you are monitoring whose performance, resilience, and security are important to your organization, is bigger than any one processor that runs any individual part of it. So monitoring a server, or watching a network address, conveys far less relevant information for distributed systems and microservices than it did before.

The challenge this fact creates is this: You need a monitoring strategy before you choose the tool that can help your organization execute this strategy. Kubernetes is not self-monitoring, nor does it include the tool you need.

Monitoring Kubernetes requires solving many of the same challenges that need to be solved with any highly scalable elastic application, though the tooling or approaches may be different. All of the Kubernetes components — container, pod, node and cluster — must be covered in the monitoring operation. Equally important, processes must be in place to assimilate the

results of monitoring and take appropriate corrective measures in response. This last item is often overlooked by DevOps teams.

Choosing a monitoring toolset is certainly important, but not for the reason you might be thinking. Every monitoring toolset has its pros and cons and, shall we say, unique qualities. You may find yourself choosing a combination of toolsets, just as Kenzan has done internally, especially when you need to monitor several facets simultaneously. The fact is, the most important thing about a toolset is that you stick to the set you've chosen, and use it consistently for your Kubernetes clusters.

We had to find that fact out for ourselves. Although there are many viable options, here are the monitoring toolsets that are frequently used by Kubernetes users, including those used by our team, and which we recommend for your organization:

- **[Heapster](#)**: Installed as a pod inside of Kubernetes, it gathers data and events from the containers and pods within the cluster.
- **[Prometheus](#)**: Open source Cloud Native Computing Foundation (CNCF) project that offers powerful querying capabilities, visualization and alerting.
- **[Grafana](#)**: Used in conjunction with Heapster for visualizing data within your Kubernetes environment.
- **[InfluxDB](#)**: A highly-available database platform that stores the data captured by all the Heapster pods.

Monitoring tools need to be just as durable, if not more durable than, your application as a whole. Nothing is more frustrating than an outage that causes your monitoring tools to go dark, leaving you without insight at the time you need it most. While best practices for monitoring at this level tend to be very specific to the application, you should look at the failure

points within your infrastructure and ensure that any outages that could happen would not cause monitoring blind spots.

Most third-party and add-on applications that monitor Kubernetes (e.g., [cAdvisor](#), Heapster) will be deployable inside your environment. Still, make sure either that logging of those applications happens outside of the cluster, or that they are set up with failover capability themselves. It's remarkable how frequently this simple but critical concept is overlooked.

Monitoring Containers

Containers are the lowest-level entity within a Kubernetes ecosystem. Monitoring at this level is necessary not only for the health of the containerized application, but also to ensure scaling is happening properly. Most metrics provided by Docker can be used for monitoring Docker containers, and you can also leverage many traditional monitoring tools (e.g., [Datadog](#), Heapster). Kenzan tends to focus on the lowest-level data to help determine the health of each individual container; for instance:

1. **CPU utilization:** Rendered as an average per minute, hour or day.
2. **Memory utilization:** Rendered as an average of usage/limit per minute, hour or day.
3. **Network I/O:** Determines any major latencies in the network, as oftentimes the effects of traffic spikes may be amplified by network latencies. Monitoring I/O may also expose opportunities for better caching or circuit-breaking within the application.

These three categories will reveal whether containers are getting stressed, where the latency is at the container level, and whether scaling is happening when needed.

NOTE: [Linkerd](#) is a service discovery proxy from the CNCF, which manages the Kubernetes project. Utilizing Linkerd as your load balancer for network traffic is a great way to implement such network management tricks as load shedding, and also for handling performance problems on account of retry spikes or a noisy neighbor.

Monitoring Pods

Pods are Kubernetes' abstraction layer around the container. This is the layer that Kubernetes will scale and heal for itself. Pods come in and out of existence at regular intervals. The type of monitoring that you may find most useful at the pod level involves the life cycle events for those pods. The data you can harvest from such monitoring may be very useful in understanding whether you are scaling properly, whether spikes are being handled, and whether pods are failing over but correcting themselves.

Lots of data may be captured from Kubernetes; this [list on GitHub](#) will provide you with complete descriptions. Here's what Kenzan's experience tells its teams to focus on:

- **Scale events** take place when pods are created and come into existence. These events give a higher-level view of how applications are handling scale.
- **Pod terminations** are useful for knowing which pods were killed and why. Sometimes terminations are on account of load fluctuations, and other times Kubernetes may kill the pod if a simple health check fails. Separating the two in your mind is critical.
- **Container restarts** are important for monitoring the health of the container within the pod.
- Lengthy **boot time intervals** are common signals of an unhealthy application. Containers should spin up and out of existence very quickly.

If you keep our monitoring somewhat simple and uncluttered, then it's much easier to manage. When you need to take a deeper, but ad hoc, dive into the metrics, you can rely on custom dashboards from a service like Grafana.

Monitoring Clusters

The processes involved with monitoring a production system are either similar enough to one another, or morphing to become that way progressively, particularly with respect to what data these processes are looking for. But the rationales for monitoring these different components will vary. At the cluster level, we tend to look at the application much more holistically. We are still monitoring CPU, memory utilization, and network I/O, but with respect to how the entire application is behaving.

Often we see a cluster fail when an application scales beyond its provisioned memory and CPU allotments. An elastic application presents a unique challenge: It will continue to scale until it can't scale any more. So the only real signal you get is an out-and-out failure, often at the cluster level. For this unique reason, it's very important to keep a close watch on each cluster, looking out for the signals of cluster failure before they happen.

We do a kind of time-series analysis in which we monitor four key variables:

1. Overall cluster CPU usage.
2. CPU usage per node.
3. Overall cluster memory.
4. Memory usage per node.

Although relatively rare, CPU usage per node can reveal one node severely underperforming the others, while memory usage per node can uncover

routing problems or load sharing issues. Using time series analysis, you should be able to plot these variables on an heuristic chart.

The New Stack has come across a container-based, distributed SQL database cluster called [CrateDB](#). It's designed to leverage the principles of distributed architecture for self-healing and instantaneous failover. CrateDB's architects say it's easily manageable by Kubernetes, and recently has been adapted to store machine data — the very class of data that's used in performance monitoring, including for the Kubernetes clusters themselves, as well as their pods and their containers.

Monitoring the Network

As more and more applications are shifting towards elastic applications with microservices, it's easy to overlook the extents to which they depend upon the network to be healthy and functioning. A highly elastic microservice application on an underperforming network will never run smoothly. No amount of defensive development or auto-healing will make it run properly.

This is why we take network monitoring very seriously. Fortunately, tools such as Heapster can capture metrics on the network and its performance. While we typically find these metrics to be useful for spotting the bottlenecks, they don't go far enough in diagnosing the root cause. This requires further digging with network specific applications.

We typically like to monitor a few items, and find it useful to separate between transmitting and receiving:

- **Bytes received over network** shows the bytes over a designated time frame. We generally look for spikes in this series.
- **Bytes transmitted over network** reveals the difference between transmitted and received traffic, which can be very useful.

- **Network received errors** reveal the numbers of dropped packets or errors the network is getting over a specified duration.
- **Network transmitted errors** tells us the quantity of errors happening in transmission.

Your insight into the underpinnings of your Kubernetes environment will only be as good as your metrics. Kenzan suggests you take findings regularly, and develop an actionable plan to resolving the issues you uncover. The action plan will need to be targeted to the application, the Kubernetes environment, or the platform it is running on. Remarkably, teams tend to forget the importance of this feedback loop.

NOTE: Currently in alpha, custom metrics is being added to Kubernetes' autoscaling capabilities. This is a very exciting development as it will allow for the most granular and customizable feedback loop into autoscaling.

Automating Kubernetes at Scale

Kubernetes is not a management platform, nor should it be mistaken for one. The whole point of orchestration is to reliably enable an automated system to facilitate the deployment and management of applications at scale, without the need for human intervention at each and every step. If the tools you use with and for Kubernetes don't enable automation, then you're not truly taking advantage of the benefits of orchestration.

Logging

Any Kubernetes production environment will rely heavily on logs. We typically try to separate out platform logging from application logging. This may be done via very different tooling and applications, or even by filtering and tagging within the logs themselves. As with any distributed system, logging provides the vital evidence for accurately tracing specific

calls, even if they are on different microservices, so that a root cause may be identified.

Here are our suggestions for logging within a distributed Kubernetes environment:

- Use a single, highly available **log aggregator**, and capture data from across the entire environment in a single place.
- Create a single, common **transaction ID** across the entire end-to-end call for each specific client. This will make it much easier to trace the thread all the way to the ground.
- Ensure that **service names and applications** are being logged.
- Standardize the **logging levels** within the entire stack.
- Ensure that no **data intended to be secure** is being logged in the clear.

Besides this high-level approach to logging, you should understand how Kubernetes handles its own logging and events.

Kubernetes nodes run on a virtual Linux computing platform. Components like kubelet and Docker runtime run natively on Linux, logging onto its local file system. Linux logging is configured at different folder locations including the ubiquitous `/var/log`. The first thing an administrator should do is validate log rotations for these log files, as well as all the other miscellaneous Linux logs. Kubernetes' documentation provides good recommendations for files to [rotate](#). The logging configuration should be inspected even if you intend to replace the local logging mechanism with an alternative.

We don't recommend that you keep logs for virtual compute instances inside ephemeral cloud computing environments. Such instances can

disappear without notice. Modern logging and analytics tools provide enough context and visual aids to help operators determine what actually transpires inside large Kubernetes cluster deployments. You should use a log aggregation service to ship your logs away from the Kubernetes environment, for later review and analysis.

There are a few reliable methods for capturing Kubernetes-native Linux logs, Kubernetes container-based component logs, and all application container log data:

- Simply extend Kubernetes' existing logging capability. As logs accumulate and rotate on the nodes, you can ship them elsewhere. One popular way to do that is with a logging container whose entire purpose is to send logs to another system.
- Alternately, you can use the [Fluentd data collector](#) to transport logs to an [ELK stack \(Elasticsearch, Logstash and Kibana\)](#), or to some other log aggregation and analytics system. You could have a logging pod with a Fluentd container on every node (Kubernetes makes this easy with the concept of [DaemonSets](#)). This log shipping method makes use of the command `kubectl logs`.
- There are variations on this approach where your application containers have a logging container in the same pod, separating the application from the system logging.

Whatever method you choose, the logs do end up residing on the node at some point, and they do have to go someplace else.

Self-Healing

We believe it's next to impossible for your system to achieve high uptime rates without self-healing capability, especially with a distributed environment. Kubernetes can regularly monitor the health of pods and

containers, and take immediate actions to resolve what issues it encounters. Two of the object types that Kubernetes natively recognizes are `podstatus` and `containerstatus`. These objects may have any of several states:

- **Pending:** Kubernetes has accepted the pod, but has yet to create one or more of the container images. This includes time before being scheduled as well as time spent downloading images over the network, which could take a while.
- **Running:** Kubernetes has bound the pod to a node, and has created all of its containers. At least one of these containers is running, or is in the process of starting or restarting.
- **Succeeded:** All containers in the pod have terminated successfully, and will not be restarted.
- **Failed:** All containers in the pod have terminated, and at least one of these containers has either exited with non-zero status, or was otherwise terminated by the system.
- **Unknown:** For some reason, the state of the pod could not be obtained, typically due to an error communicating with its host.

The kubelet agent that runs on each node is capable of obtaining more detailed health checks, using any of these three different methods:

1. **ExecAction:** Runs a specific command within the container.
2. **TCPSocketAction:** Performs a simple transmission control protocol (TCP) check on a specific container to ensure its existence.
3. **HTTPGetAction:** Performs a simple HTTP GET check on the container, expecting to get a response of 200 (OK) or 400 (bad request).

A kubelet can also probe the containers within a pod for “liveness” (its ability to respond) and “readiness” (its preparedness to handle requests). You can configure a liveness check to meet your specific needs, as demonstrated by the YAML file in Example 5.1:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
    - args:
      - /server
      image: gcr.io/google_containers/liveness
livenessProbe:
  httpGet:
    # when "host" is not defined, "PodIP" will be
used
    # host: my-host
    # when "scheme" is not defined, "HTTP" scheme
will be used. Only "HTTP" and "HTTPS" are allowed
    # scheme: HTTPS
    path: /healthz
    port: 8080
    httpHeaders:
      - name: X-Custom-Header
        value: Awesome
    initialDelaySeconds: 15
```

```
timeoutSeconds: 1
name: liveness
```

EXAMPLE 5.1: *A configuration file for establishing a liveness check. [Courtesy Kubernetes.io]*

With the data returned from a config file like this, Kubernetes has the ability to create **restartPolicies**. These policies will tell Kubernetes what to do in the event that any of the health checks fail. The system will only restart at the pod level, so all containers within a pod will need to be restarted.

The specifics of your self-healing configuration will be based on your needs and those of the application. Fortunately, Kubernetes provides quite a bit of flexibility and configurability data from many different locations.

NOTE: While self-healing features such as rebooting every half-hour or so is great to have, it can also mask a problem with your application. You need monitoring and logging functions that are robust enough to bubble up any issues that may occur.

Resilience

Depending on the needs of your application (e.g., 99.999% uptime) resilience testing can and should be part of your platform. Failure at any level of your application should be recoverable, so that no one experiences any amount of downtime. In our experience, bulletproof applications are only feasible if development teams know in advance their work will be put through extensive resilience testing.

Although you can conduct a type of resilience testing through the simplest of manual methods, such as manually shutting down databases or killing pods at random, our experience has proven these methods are much more effective when they're automated. Although Netflix's [Chaos Monkey](#)

is a very powerful, tremendously useful, resilience testing tool that runs in AWS, sadly it was not built for Kubernetes. Thankfully, there are emerging resilience testing frameworks in the Kubernetes sphere, two of which are [fabric8 Chaos Monkey](#) (part of the fabric8 integrated development environment) and [kube-monkey](#).

Routine Auditing

No matter how many checks and balances you put in place, your Kubernetes production environment will benefit from routine maintenance and auditing. These audits will cover topics that normal monitoring will not cover. Traditionally, auditing is taken on as a manual process, but the automated tooling in this space is quickly and dramatically improving.

Typically, some of the issues we expect a good audit to turn up include:

- **Container security vulnerabilities:** One of the chief sources of security holes in containers is vulnerable dependencies that are included with them. Regular security patches should be made to the containers; it is important to keep them up to date. But this doesn't have to be like patching a part of a virtual machine (VM). Instead, you'll find it easier to simply rebuild vulnerable container images with updated, patched dependencies.
- **Containers running in privileged mode:** Although we've heard some reasons why containers may need to run in privileged mode, we try to avoid doing so, or only allow it for a very short period of time. A routine audit of all pods running in privileged mode is a valuable process to ensure against the building up of weak points due to unchecked privilege.
- **Containers without an owner:** Allowing a container not to have an owner in its native namespace is a potential security vulnerability.

- **Image size:** Find any images that have grown too large, then rebuild and replace them. The accrual of log files, adding state and then writing to file systems, in-memory processing operations accumulating over time, persistent states, and untidy caching processes all lead to bulky and unmanageable container images.
- **Single replicas:** Any pod having only a single replica may be a sign of a single point of failure.
- **Unapproved container repositories:** Allowing an image to be downloaded from an unapproved registry is an engraved invitation to vulnerability.
- **Invalid ingress:** For example, a “wildcard ingress” from a domain or subdomain can render a pod vulnerable.

NOTE: Another way to protect against introducing a single point of failure is to ensure your master node is replicated.

Scaling Clusters

For Kubernetes, scaling can mean any of three things: making each node in a cluster larger (increasing its compute, memory and storage), adding more nodes to a cluster, or adding more clusters. The good news is, the vast majority of scaling is handled by the Kubernetes scheduler. It will not schedule any pod if the resources for it are not available. This means the environment won't crash, though it also means users may experience latency or downtime.

In a Kubernetes environment, a replication controller continually monitors the desired state of a system (as determined by its configuration) and makes changes to the system's current state so that it more closely matches the desired state. It's the replication controller that defines the number of pod replicas that should be running at any given time. That

number is essentially the base level for each pod. Allowing Kubernetes to manage pod instances through a replication controller is one way of ensuring that all pods are running and healthy.

[Horizontal Pod Autoscaling](#) is a tool Kubernetes uses to automatically scale the number of pods in a replication controller, based on CPU utilization. The ability to scale using application-supplied metrics was in alpha at the time of this writing. With horizontal autoscaling, when resources are getting over-utilized, Kubernetes will pick up on the spike and throw more pods at the load. Generally, Kubernetes adds enough pods to handle the spike.

Scaling at the Infrastructure as a Service (IaaS) or data center level typically involves allocating more instances and more memory to the environment, and creating more nodes. As soon as the nodes are available, the scheduler will begin to schedule pods to that node.

Resource Quotas

A resource quota lets you limit a namespace within your Kubernetes platform, ensuring that one application will not consume all the resources and impact other applications, as demonstrated by the command-line interaction in Example 5.2.

```
$ kubectl describe quota compute-resources
--namespace=quota-example
Name:                                compute-resources
Namespace:                           quota-example
Resource                             Used  Hard
-----
limits.cpu                           0      2
limits.memory                         0      2Gi
pods                                  0      4
```

<code>requests.cpu</code>	<code>0</code>	<code>1</code>
<code>requests.memory</code>	<code>0</code>	<code>1Gi</code>

EXAMPLE 5.2: *A sample resource quota, returned by the `kubectl describe quota` command.*

Setting resource quotas can be a bit challenging. In our experience, we’ve found breaking down the namespaces by their expected load and using a ratio to calculate percentage for the cluster is the most diplomatic way to begin. From there, use monitoring and auditing to determine if your partitioning is correct.

For example, suppose you have a cluster with 16 GB of RAM, eight virtual CPUs (vCPUs), and three namespaces. You would need to set at least 15 percent of main memory (about 3 GB) for system daemons such as kernel and kubelet, and allocate a load breakdown with the remaining 13 GB. You could then break down the load percentage into rough quarters, with Namespace A given 50 percent of the load and four vCPUs, and Namespaces B and C each getting 25 percent of the load and two vCPUs.

Container Resource Constraints

Figuring out how much resources an individual container or pod will require has become something of an art. Historically, developer teams have made their estimates way more powerful than they need to be. We try to perform some level of load testing to see how it fails over, and

Typical Cluster Allocation Breakdown			
Namespace	Load Percentage	vCPU	Memory
Namespace A	50%	4	8 GB
Namespace B	25%	2	3 GB
Namespace C	25%	2	2 GB

TABLE 5.1: *Determining scalable resource allocation for load, vCPUs, and memory in clusters of containers.*

then allocate resources appropriately. Netflix coined this method “squeeze testing.”

Through a YAML configuration file, a Kubernetes operator presents the orchestrator with “bids” for available resources, although that’s not the official name for it. Example 5.3 shows a sample config for a container compute resource, borrowed from [Kubernetes’ own open source examples](#). This configuration defines the pod as having two separate containers, **mysql** and **wordpress**. Both the containers in this pod make declarations requesting essentially 64 [mebibytes](#) (MiB) — essentially megabytes, but in powers of two rather than ten — and 250 millicores (m) for one-quarter of a vCPU. These requests are for general requirements, though it’s conceivable that the orchestrator could exceed those requests. So, as this example shows, both containers in the pod set limits of 128 MiB and 500 millicores (half a vCPU).

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
```

```
        cpu: "500m"
-   name: wp
    image: wordpress
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

EXAMPLE 5.3: *A sample configuration for a pod that accounts for both MySQL and WordPress.*

Setting up the proper memory for applications has become both an art and a science. A proper allocation should take account of the sizing needs of applications. We typically factor in a couple things:

- **Application volatility:** If you are expecting large fluctuations in the number of users or in the overall application demand, then having more space to expand is a good thing.
- **High availability requirements:** If you truly need high availability and are spreading workloads across two locations (e.g., AWS regions), then should one location go down, you want to ensure that the single remaining location can handle the full load.

Typically, we shoot for the 60/40 rule. We traditionally will allow 60 percent of resources to be allocated at any given time, leaving 40 percent of free space for growth. With monitoring, we can watch for resource consumption spiking too close to the upper limit, and add more nodes to compensate if it does.

Facilitating Kubernetes and Applications

There are two schools of thought with respect to the relationship between the containerized application and the container orchestration system.

One is that the developer should be enabled to take a completely agnostic attitude toward the platform or platforms on which the application will be deployed. This is one of the hallmarks of serverless architecture — abstracting away the details of configuration and deployment from the developer — which Kubernetes certainly supports.

On the opposite side of the aisle comes this argument: The best way for developers to build applications for the platforms their organizations will use is to enable those applications to detect their own configurations by means of APIs, and adapt themselves dynamically to the performance requirements of whatever infrastructure happens to be running them at the time. Kubernetes also supports this school of thought.

This is one of those areas where Kubernetes' contributing engineers declare their platform unopinionated. But this is not to say that the orchestrator has, or should have, no direct relationship with the applications or workloads contained in its pods, whichever methodology your organization may choose to adopt.

Mutual Requirements

A healthy Kubernetes environment will have a steady, bi-directional flow of information from the platform into the application, and from the application back into the platform. This means these two components really do need to look out for one another. Despite the layers of architectural abstractions separating the application from its host, now that easily distributed applications have become reality, applications created during the era of distributed architecture need to be aware of distributed platforms.

Conversely, platforms such as Kubernetes should never treat applications as amorphous entities that all fall within strict quotas for resources and connectivity. You might think this is one of those assertions that goes without saying, but you'd be surprised how little bi-directional communication takes place today between the two disciplines.

- **Availability:** Not every application needs 99.999 percent availability, but some definitely do. There are costs and challenges associated with the level of availability that is required. Your application staging platform should factor in the needs of each application. If a high level of availability is required, it should ensure that all resources made available to the application are redundant — including data, load balancing and service discovery.
- **Scalability:** It's far too late in the process if you're just beginning to assess the scalability needs of your application right before it heads to production. Your development team may never have considered it, so quite possibly the application can't even handle it. Taking account of scalability needs will enable you to set a baseline for services, and a threshold for each individual pod. These metrics will prove crucial for monitoring memory and CPU usage.
- **Container image sizing:** Container images come in many different sizes. The services included in those containers may have very different needs. Knowing exactly how the services will be used and where they will start to churn (for instance, memory, CPU, network) should make it easier for you to size the container properly, and get the most out of the base sizing.

A healthy Kubernetes installation will definitely establish a baseline of patterns and anti-patterns to which your applications should adhere. This

will create consistency, and ensure that monitoring focuses on the right metrics when optimizing the application. We typically come across a few key points where the platform impacts the design of applications.

- **Bootstrapping requirements:** Very poorly written pods will not start up very quickly. Slow startups should never be allowed in a Kubernetes production environment. Given the platform's elastic nature and continuous monitoring, slow applications can dramatically impact its health. We typically don't permit applications to require more than 20 seconds for startup. This rule forces developers to maintain lightweight services, and make sure dependencies are kept to a minimum.
- **On-pod needs vs. off-pod needs:** With Kubernetes, you may find it very convenient to embed smaller containers (called supporting containers) into the same pod as the application. These tend to be very small and lightweight, and handle a lot of the platform-related code such as logging, secrets and configuration.
- **Ephemeral caching:** Every well-performing application will have some level of caching. You'll want to standardize how that cache is managed, so that the platform can support distributing the cache if it needs to, as well as monitoring any persistence layers the application may require when a pod is terminated.
- **Application logging:** We've used many different log platforms — for instance, [ELK stack](#), [Splunk](#), [Graylog](#), [Datadog](#) — and you can be successful with all of them. It should be the responsibility of your platform team to dictate how logging should occur, and to set your developers' expectations for capturing application-level metrics and events, such as HTTP errors, stack errors, resource usage levels and transaction IDs.

Configuration Management

Every application's configuration is a principal driver of its behavior. With Kubernetes, all applications and pods should be handling configuration consistently. Otherwise, achieving rapid deployments will be very difficult, and ensuring accuracy of configuration will be cumbersome.

A common approach that we see in managing config is to use an “externalized configuration” service. There are existing microservices patterns and solutions that allow for externalized configuration, such as [Spring Cloud Config](#), in addition to simple key-value stores. Such a configuration service is typically its own pod with a data store backing it, typically off-pod.

Kubernetes provides other ways of managing configuration that are inherent to the platform. The two most commonly used methods are:

- **Environment variables** for specific container-level parameters. These are set directly in the container, and are specified in the pod-level YAML configurations.
- **Config maps** are similar to environment variables as they are container-level, but have more complex data structures such as key-value pairs.

There really is no correct way to handle configuration other than to have consistency. We typically use environment variables and config maps to manage the platform-level configuration, and the externalized configuration service to manage application-level config.

So it is imperative that your application developers are taking direct feedback from the needs of the platform, and that your platform immediately benefits from the changes they make. This is really the only way you'll have applications that are robust enough to grow and evolve

within a Kubernetes environment. Don't get too stuck in the right way, but do have a consistent way.

Enabling Statefulness

Introduced into beta for Kubernetes version 1.7 (and thus perhaps not ready for prime time everywhere), a stateful set (referenced using the `StatefulSet` object) can provide ordered, stable, graceful and generally nice storage, network configuration and deployments — essentially kinder, gentler pod management. This is important for applications that need their services started and stopped in a predictable manner.

For example, an application stack may prefer to have data generating and mutation services stopped first on shutdown and started last on start-up. Replicated stateful sets leverage this order guarantee to allow for supporting data sharing scenarios, such as master data repository containers with read replica containers. This enables a complete stateful, data persistent application to reside wholly within the context of Kubernetes.

NOTE: An organization should practice and simulate start-up, shutdown and failure scenarios before using these features in production. This is for Kubernetes operational confidence, but also for the applications.

Replicated StatefulSets should be considered an advanced capability.

But before you consider implementing everything that your application needs to remain stateful in a Kubernetes stateful set, you should make a complete cost-of-ownership evaluation, taking into account the resources available to you through a public IaaS. Essentially, stateful sets are really powerful tools, but should be utilized carefully and judiciously.

For instance, you may use a stateful set for an relational database management system (RDBMS), but your IaaS offers a fully-managed RDBMS solution — one that might even be able to handle, or at least can

geographically replicate, the global data consistency problem. To that end, some public IaaS platforms even offer a RDBMS proxy container, or perhaps the relational database service vendor may offer one. This is a useful way to fully isolate and maintain the datastore dependencies outside of your application code, thus reducing your dependency on stateful sets.

Coordinating the Deployment

Organizations typically separate software development teams from systems operations teams, so that responsibility for active code shifts. Because of this, it becomes incumbent upon the software platform itself to enforce standards for the code it supports. No code should be deployed in Kubernetes without meeting minimum requirements.

This means someone should set those minimum requirements for deployments. Here are some recommendations:

Small Footprint and Bootstrap Times

No pod should consume too many resources. This is a sign of either a poor solution architecture (e.g., too monolithic), or poorly written code. We recommend you limit your memory maximum to 2 GB, and bootstrap time to under 10 seconds.

You could have great custom, distributed application code, but if your data team decides to bake the database directly into the pod, the greatest code in the world won't save you from slow startup times.

Adequate Automated Test Coverage

The meaning of adequate may be a bit subjective, and may vary in accordance with the requirements of the workloads you run. Perhaps not all of your tests can, or should, be automated. But it's critically important that automated testing play some role in your deployment pipeline. Specifically, your pipeline should be capable of running a full and exclusive

suite of tests for each pod deployed, as soon as possible. Each test should be capable of failing the deployment if its requirements are not met. We rely heavily upon these types of tests:

- **Unit tests** should run in the build process and are applicable to the smallest chunks of code. Here, you don't test all the code. You wouldn't even test most of the code. You would test only the impacted code. Implementing unit tests starts with running them as part of the build process. Later, you extend them to integration tests for all dependencies and consumers, functional testing to regression test dependencies, and finally lightweight, targeted end-to-end (E2E) tests. This allows for higher confidence in smaller releases.
- **Integration tests** pertain to the testing of all the functions and modules, ensuring that they are working as expected. This is probably the most misunderstood type of testing in our experience, especially in microservices environments. For us, integration testing is about integrating functions and modules. It is not about integrating microservices — that comes from functional testing. Contract testing (e.g., [Pact](#)) is a really powerful style of integration testing, as it allows for very targeted tests directly to the code modules being deployed.
- **Functional tests** ensure that all the dependent components (through the soon-to-be-deployed pod) are functioning as expected. This is done by testing the inputs and asserting on the outputs. This is the first time that testing can happen with all the upstream dependencies and — at least theoretically — the downstream dependencies in place.
- **End-to-end tests** mimic how a user will interact with the workload in a production deployment. These tests tend to have the lightest weight, and are smoke tests as much as anything else.

Automated Deploys

Manual deployments are generally dangerous, and rarely yield consistent results. Kenzan typically deploys workloads using automated platforms such as [Spinnaker](#) and [Jenkins](#). Through a technique called a canary release, a workload slowly rolls out into production over several hours' time, with its error rates continually monitored. If error rates are consistently high, this process can trigger an automated rollback. You should avoid building any system that requires manual intervention to roll back.

Conclusion

Choosing Kubernetes as your container orchestration solution solves many of the container management problems with which organizations are faced today. This chapter highlighted the many deployment and operational concerns organizations face when operating Kubernetes in production. It then addressed how using Kubernetes enables microservice application architectures. For containerized, cloud-native apps, Kubernetes provides almost Platform as a Service (PaaS)-like capabilities beyond just container management, coupled with great operational resilience.

MAINTAINING THE KUBERNETES LIFE CYCLE



Kubernetes now faces an evolutionary hurdle that only Docker before it has faced so soon in its life cycle: the problem of multiple concurrent versions. Version 1.7 is upon us today, even though other versions are still in active use. At the rate Kubernetes is evolving, over a dozen viable versions could be deployed over the standard life cycle of organizations' IT projects — some three to four years' time.

“We try to make every new Kubernetes release as stable as possible,” said Caleb Miles, who leads the Kubernetes Project Management SIG. “We have a long code freeze and stabilization period to shake out any of the rough edges before a new version of Kubernetes lands.”

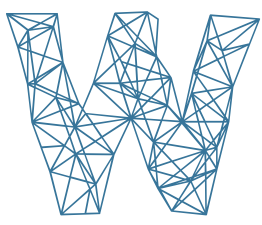
In this podcast, learn more about how the Project Management SIG maintains consistency and concurrency without introducing obsolescence. [Listen to the Podcast.](#)



Caleb Miles technical program manager at CoreOS, is helping to support the Kubernetes open source community through a focus on contributor experience, the release process, and project management. Prior to CoreOS, Caleb worked on Cloud Foundry at Pivotal Software, where he focused on maintaining and improving infrastructure support for the Cloud Foundry platform. Previously, Caleb contributed to the Ceph distributed storage system, working on the S3 compatible object storage interface.

ROADMAP FOR THE FUTURE OF KUBERNETES

by **SCOTT M. FULTON III**



We've been asking the same questions about our computing platforms ever since we began excavating the basement floors of our headquarters buildings, making room for the likes of IBM, RCA, Sperry and DEC: What happens when these machines reach the end of their life cycles? Where do their programs go after that time? And will the new machines that replace the old ones run the old programs the same way?

We didn't call it infrastructure at the beginning, even though it shared the basement with the furnaces and the laundry. Sometime during the 1980s, academics who joined the British Government began using that term to describe information technology as a general commodity. The notion emerged so gradually into the mainstream conscience that not even historians can pin down the exact date when the metaphor was first uttered in public.

Over time, their metaphor congealed into a kind of lexical platform. The Information Technology Infrastructure Library (ITIL) codified in documents, perhaps for the first time, the primordial ideas upon which

the subject of this book is based: the delivery of technology as services, the management of processes as pipelines, the institution of continuous improvement methods.

Delivering Infrastructure as a Service (IaaS) was not some radical discovery that somebody introduced to the world in a TED Talk, and the world embraced like a viral cat GIF. IaaS was something we were already doing, and the revolution was done before we realized it.

“I think orchestration tools are really fascinating and super-awesome,” said [Laura Frank](#), director of engineering for continuous integration platform maker Codeship. “And because they are fascinating and super-awesome, and the cool new thing, engineers — as maybe a character flaw of our profession — really gravitate toward making things complicated. We want to check out all the new cool things and find an excuse to use it.

“I think a lot of times, engineering teams choose to use a tool that might be more complex for their actual needs,” Frank continued, “and they try to solve problems that they don’t quite have yet. I don’t think that’s necessarily specific to engineers; I think lots of people tend to do that. Maybe it’s just a little more pronounced with engineers.”

Thanks to the dreams and intentions of engineers, however complex they may be, in nearly seven decades’ time, computing has become the cloud beneath us. We only think we use supercomputers in our pockets, but the truth is that the real work is being done in cloud data centers. They may still be our basement floors, or they may be leased space from Equinix or Digital Realty, or they may be virtual constructs supported by AWS US-West-1 or -2. But they are the true seats of power in our world. Our smartphones and tablets are just putting on a good show for us, but Siri and Spotify and Slack and Salesforce have made their homes in the places their predecessors left for them.

“If we’re going to survive as a project ... we have to allow for flexibility and heterogeneity and openness ... in order for us to not become stuck in what we have done before.”

-Brendan Burns, Microsoft.

So little has actually changed.

The questions we ask about the new computing platforms (new for us, or so it seems) in which we make our investments are no less pertinent today for virtual platforms, such as vSphere, Mesosphere, OpenStack and Kubernetes, than to the physical platforms of the 1950s — the RCA 501, the IBM 701 — or the back-office power centers of the 1980s — the IBM AS/400, the DEC VAX-11. The platform life cycle is the critical subject every institution asks of the analysts it hires to investigate a potential new platform. How far into the future, businesses have always asked, will this platform take us?

What actually has changed — both suddenly and radically — is the context of our answers. The explanations for how and why things in computing infrastructure work today are so different than they were just three years ago, let alone the past seven decades of the computing industry, that at first glance they may as well have come from an alien planet.

Who Makes Kubernetes, Really?

“Tools should be selected in order to deliver value to the operator of the

tools,” said CoreOS’ Caleb Miles. “You see some organizations find that Kubernetes by itself delivers important value to their organization and to their operations. We see that other organizations combine upstream Kubernetes with a variety of other tools and projects to deliver value to their organizations.

“I believe that the CNCF serves an important function,” Miles continued, “as a clearinghouse for helping organizations on their cloud-native journeys — to figure out what tools can be used to solve their challenges. [In] improving developer productivity, increasing developer velocity, improving transparency into operations — I think the CNCF can be a real partner to help organizations and users make that journey.”

Kubernetes, described Google’s [Tim Hockin](#), “is over a million lines of code that spans dozens of repos, that has hundreds of active developers on it every week.” He is a principal software engineer at Google, and Kubernetes’ co-founder. “There’s simply no way that one person can read even the subject of every pull request and every bug that comes through, much less spend time actually analyzing and participating in all of those issues.”

Hockin is known for his contributions to Borg, the internal container orchestration system at Google that revolutionized its approach to deploying applications, and that became the first prototype for Kubernetes. Combined with his contributions to Docker, he became a recognizable influence in popularizing distributed containerization. However, it’s not a perception that Hockin would readily accept, citing the complexity of the system that has emerged around containerization, and the existence of many other influencers the development community.

“I think it’s fair to call it a ‘Kubernetes galaxy,’” said [Aparna Sinha](#), Google’s group product manager for Kubernetes, “because it is a very large project. By many measures, it’s one of the largest projects on GitHub.”

Kubernetes is an open source project of the Cloud Native Computing Foundation (CNCF), which is itself part of The Linux Foundation. By design, the CNCF's leadership is not led or maintained by any one company. While Google engineers played a major role in the group's formation, Microsoft joined the CNCF in June 2017 and became a platinum-level member in July. Cisco, Dell Technologies, Huawei, IBM, Intel, Red Hat, CoreOS and Samsung are also members at the highest, platinum level.

“For new contributors, it's a little bit overwhelming at first to look at the list of SIGs [Special Interest Groups] — there are twenty-plus SIGs — and try to figure out, ‘I have a problem, I have a question, where does it belong?’” admitted Hockin. “But I think we've done a reasonable job at offering guidance on that. And as the governance structure formalizes, each of these SIGs will have a more formal charter, which details in a little bit more depth what the SIG is responsible for, where its boundaries are, and what touchpoints it has with other SIGs.”

Where is the Center of Power?

Ostensibly, this SIG-based structure intentionally guards against any one vendor's control over the entire project. However, many folks in today's IT leadership — veterans of decades of Windows patches and years of golden master virtual machines — are understandably skeptical about whether this “development community” is really a nicer-sounding moniker for a “development committee.”

But the CNCF is not a United Nations, or a board of trustees, for these member companies. Instead, the organization seeks to identify the interests of the users of Kubernetes, and of the other projects it manages that are compatible with Kubernetes. Granted, those users are typically customers, or potential ones. Yet identifying users first and foremost enables the CNCF to drive a development strategy for the platform whose

objective is to respond to the needs of those identified customers, if not anticipate what those needs might be in the future.

That's a very different strategy than one whose principal motivation is softening the targeted market to be more receptive to whatever shape, form or function the platform happens to take at the time. How often have we seen that before?

“Not everyone is ready to deploy 12-factor, cloud-native applications today. We try to make the new version of Kubernetes worth it for people to adopt.”

-Caleb Miles, CoreOS.

SIGs, explained Sinha, “are working groups composed of engineers and cross-functional individuals from many different companies, as well as independent individuals, who have come together around a topic of interest. Each SIG develops a roadmap for that SIG for the year. And I think that's part of the reason why it can sometimes be hard to see the themes that are emerging at the project-wide level.”

The roadmap for each SIG serves as a combination of an itinerary and a constitution. It explains its function to the world by presenting itself in terms of what it plans to accomplish.

“So you can consider each of those SIGs as its own planet or solar system,” said Sinha.

Google's Tim Hockin refrains from accepting any sort of central

authoritative role in the project, telling us he's quite comfortable with allowing responsibility for certain issues — around which individual SIGs are formed — to be delegated to others. “I don't need to be involved in every issue at this point,” said Hockin. “The team of developers here at Google and across the world are very well trusted, and I believe they're doing the right things even if I'm not paying attention.

“The size of this project requires that I choose the things that I pay attention to,” he continued, “which is just different for me because I like to be involved in everything, and I like to understand how things are evolving, and what decisions people are making and why. Necessarily, as a project grows up, that just can't be true anymore.”

“We can talk about tools, and we can talk about frameworks, and delivery pipelines all day long. But really, when it comes to high-performing engineering organizations,” said Codeship's Laura Frank, “I think what everything comes back to, is developer economy. Giving engineers more ownership, or a feeling of ownership, over the code that they're writing and shipping is paramount. It's critical, when you want to talk about developer happiness, productivity, investment in the code, making sure it's the best it can be.

“For a long time, when people were very intimidated by the complexity of Kubernetes,” Frank continued, “I always encouraged them to think about where Kubernetes came from, why it exists and what kinds of problems it was trying to solve. And I often said Kubernetes is great if you have Google-sized problems, or Netflix-sized problems, or any other kinds of large application that has huge scaling problems. It's likely that you want to think you have that size of problem, but maybe you don't. It's always better to start a little bit smaller and work your way up.”

Interchangeable Parts

It is a perplexing thing: an organization whose natural leaders — or, more fairly, the people most likely to be nominated as such — may not have the personal bandwidth to assume a central leadership role. To understand how such an entity can possibly function, you have to do perhaps the most unnatural thing an analyst of a computing platform for an enterprise might expect to do: decompose each element of the platform into self-contained components that either may interact with one another, or be replaced by other components that may interact better. Kubernetes has become a platform of interchangeable building blocks, whose ultimate structure is itself a building block of a system on a different level: the growing list of CNCF hosted projects, which some would say resembles a stack.

“We’ve already started breaking up the monolithic [kubernetes/kubernetes repo](https://github.com/kubernetes/kubernetes),” Hockin explained further, referring to the GitHub location for the central repository. “We’ve been injecting stand-alone things into their own repositories; we’ve got some new rules around what goes into the main Kubernetes repo and what doesn’t; we’ve set up, through some architecture diagrams and documentation, a layering structure of what’s part of the core and what’s part of the ecosystem. And that helps guide decisions about, how do we modularize and delegate.”

The orchestrator was supposed to have been a system of interchangeable building blocks. And so is the full set of CNCF projects, at another level. But the instinct to coalesce ideas into a single, rational construct can be so great that, at times, the architects of Kubernetes do seem to be counteracting the pull of gravity.

“Kubernetes, from the beginning,” Hockin continued, “was many components working together. So it’s not like it’s a necessarily monolithic

behemoth. We already have five or six main components that are really conceptually different, while having some amount of overlap — and there's some value to developing and testing them together. And they have different teams of people working on them that are reflected in our SIG structure.”

Hockin said he hates silos. Many people who live and work in organizations that continue to compartmentalize their work processes say they hate silos. There's a complete division of labor which, it would appear from Hockin's explanations, he would rather the SIGs avoid. But some division is necessary, he believes, in order for developers and contributors to be able, during some portion of their days, to focus on achievable tasks at hand. His metaphor for this kind of division is “soft fencing.”

Keep in mind, some of Kubernetes' contributors are not engineers with a major vendor, service provider, or telco, but developers employed by firms to build their web sites and get their IT assets ported to that big cloud where all the big data lives. Some of the most important work done on this platform is contributed by volunteers.

“A number of security vendors with a networking focus got together and said, ‘Look, we need to fix this, but we need to do it in an interoperable way.’”

-Gabe Monroy, Microsoft.

So the Kubernetes project governance structure, stated Hockin, codifies the functions that each group performs as a group, rather than stipulating

the job function of each individual member. SIGs have responsibilities and ownership, and they do have technical leads. Only now is the process of determining who those leads should be becoming formalized.

“The structure is really for the benefit of developers and project maintainers,” said Hockin, “so that we can understand where bugs go, and where changes impact, and who needs to coordinate with whom. I wouldn’t expect an end user — somebody who doesn’t get involved with the community — to rationalize the structure here. That’s why we have the main `kubernetes/kubernetes` repo — the central starting point for anybody who wants to file a bug, ask a question or comprehend something.”

End user contributions like these, he went on, get directed to the appropriate SIGs and mailing lists — and it’s through this form of disseminative interaction that the Kubernetes contributors know what their users, implementers and customers expect of them.

Unlike the case with an operating system or a web browser, an infrastructure platform deployed in a data center can’t just upgrade itself automatically with absolutely assured safety — not without jeopardizing the stability of the services that are staged on that platform. So the new release of Kubernetes cannot immediately render the old releases obsolete — or, as an operating system vendor would have it, automatically vulnerable.

Caleb Miles, a technical program manager with CoreOS, is one of the leads for the Kubernetes Project Management SIG, and also a key member of an emerging group called [kubernetes/sig-release](https://github.com/kubernetes/sig-release). Miles’ function with the project would seem unusual in another era: He’s helping to ensure the stability of emerging and upcoming releases of the platform (at the time of this writing, Kubernetes was on version 1.7.3) while actually maintaining

the integrity of previous releases going forward.

“We have a long code freeze and stabilization period that we try to use to shake out any of the rough edges before a new version of Kubernetes lands,” said Miles. “But we also want to make each release important enough for companies, users of Kubernetes, and the open source community, to want to adopt the new version. Whether that’s migrating important API endpoints and resources from alpha to beta to GA (general availability), or addressing critical gaps in the project that make it difficult to deploy the workloads that the users are trying to adopt today ... Not everyone is ready to deploy 12-factor, cloud-native applications today. We try to make the new version of Kubernetes worth it for people to adopt.”

Which Transition Takes Precedent?

The challenge with achieving this goal is that the needs and requirements of businesses are actually changing more rapidly than technology. Many know their systems are not keeping up with demands. If anyone needs to make a change, business leaders believe, it’s the IT department.

“What we’re seeing from customers is that the IT conversation has shifted,” explained [Brian Gracely](#), Red Hat’s director of product strategy for OpenShift (its commercial Kubernetes platform). “People [used to] say, ‘Look, I need something to perform faster, or I need it to be cheaper.’ It was either cost-driven or CPU-driven. And with the conversations we’re having these days, the technologists in the room — whether they’re part of IT or a line of business — are saying, ‘Here’s how our business is changing. Here’s how some part of our typical go-to-market, or our supply chain, or our customer interaction is significantly changing ... I’m not going to be able to keep doing what I’ve done for the last ten years the same way.’”

It’s not so much that the supply chain is evolving or even mutating, Gracely

believes. It has actually already transformed, as though someone in the control room flipped the switch after all but forgot to send an email to IT.

“Instead of everything running in one monolithic application for your supply chain,” said Gracely, “the things that make up the service you deliver come from a bunch of different pieces and parts. And that supply chain is APIs [application programming interfaces].”

From a software developer’s perspective, an API is an endpoint for connecting services. The contract with which services are exchanged over that endpoint is the interface. From a business analyst’s perspective, an API is the connection point between elements in the supply chain. It makes tremendously greater sense to have software adapt itself to individual services, and facilitate the APIs that are already emerging in business.

The digital transformation that is truly necessary here is for even the most innovative software in use today to quite literally get with the program. It is actually Kubernetes that is in a race against time to keep up with the transformation going on upstairs.

“If you look at our overall themes for 2017, one of the big themes is to scale the project,” explained Google’s Aparna Sinha. “That’s kind of a table-stakes item, but we have to make sure that it’s easy for contributors to contribute to the project; we have to make sure that the project is architected in way that there are multiple layers; and that the core or nucleus of the project continues to get more and more stable, so that our users can rely on it.”

Who or What Provides Kubernetes with Security?

The stability of the core of the Kubernetes project will be the source of its

reliability and resilience. If developers build the functions at the periphery upon a stable core, then those functions can draw upon that core for their support.

It's a nice ideal, but here is a principle proven both by observation and history: Platforms whose foundations are prone to expansion tend to lose the interconnectedness of the functions that rely on them, and thus put themselves at risk of losing their integrity. It's the reason, at a very deep level, why the PC industry is at a standstill and the mobile and cloud computing industries are not.

“People find value in the platform, in order to deliver value to their users — in the same way that Linux is ubiquitous, but not talked about much at the distribution level anymore.”

-Caleb Miles, CoreOS.

“I think we have an obligation to provide users with a secure environment,” stated [Brendan Burns](#), one of Kubernetes' lead engineers, now a partner architect at Microsoft. “But at some level, security is a continuum. We want to make sure people can't shoot themselves in the foot. But we also understand, there's going to be a continuum depending on the kinds of data that you're using, and there are going to always be add-on tools that some people are going to want to use, add-on configurations.

“I think we do want to make sure that, when there are things like security policy and other APIs that users are going to use, we want to make sure

those APIs work well in the services that we're deploying and providing our customers with," Burns continued.

The Platform People See

It's a message with a deeper implication: Kubernetes will continue to surface itself in a variety of customer-facing platforms with big vendor brands. It will be these vendors that present the orchestrator with the more conventional image enterprise IT decision makers are looking for, along with the single source of support. But it may also be these vendors that provide the system with the security methods necessary for it to become adopted by enterprises, while maintaining compliance with the policy-driven mechanisms they already have in place.

"Kubernetes as a platform allows for many other systems to integrate with it," explained CoreOS' Caleb Miles. Miles cited [GitLab](#) open source code review project as one example of tight Kubernetes integration, "to provide more of a higher-level platform experience for development teams using GitLab.

"So the Kubernetes platform tries to be rather unopinionated about how other projects interact with Kubernetes," he continued. "Application developers can push their code directly to Git; GitLab picks up those changes, can run the tests defined by the developers; and then after running those tests, can deploy those applications directly to Kubernetes."

The mental picture Miles draws depicts a system that integrates Kubernetes so tightly that the application utilizing it may as well have consumed it, rather than the other way around. "This gives third-party platforms, tools and vendors an opportunity to build a rich ecosystem on top of Kubernetes that is tailored to the experiences expected by their user base," he said.

GitLab absorbs Kubernetes as the interchangeable orchestration engine of its continuous integration / continuous deployment (CI/CD) system. Now, Kubernetes engineers are starting to paint a picture of a security system

that absorbs the orchestrator in a similar way, specifically with respect to implementing existing network security policies.

The Policy People Trust

Although its existence pre-dates the turn of the century, policy-driven security is, for a great many enterprises, a very new thing. But it is not as new as Kubernetes — which, for those whose IT decision makers are aware of its existence, may as well have happened last week. This new wave of security is modeled on the hypervisor's capacity to serve as a gateway, protecting processors from executing damaging or malicious code from first-generation virtual machines. Replacing virtual machines (VMs) with Docker-style containers this early in the security system's life cycle is not an option for them.

While staging Kubernetes environments inside VMs may protect these environments the same way hypervisor-based security protects applications in VMs, this also severely constrains organizations' options for scalability and cross-cloud deployment — two of the big virtues of Kubernetes in the first place.

“Kubernetes, from the beginning, was many components working together. So it's not like it's a necessarily monolithic behemoth.”

-Tim Hockin, Google.

In the current Kubernetes structure, who or what determines the securability of the platform's features and functions? Or is there a means within the individual SIGs to delegate ownership of security matters to particular people?

“I would say the first strategy,” responded Google vice president of infrastructure [Eric Brewer](#), “is to try to make it so that, in most of these places, the security is not in scope.”

What’s that supposed to mean? “We’re assuming, for example, that you’re running inside a VM if you need to,” said Brewer. “And that takes care of a certain range of attacks. There are other places — one obvious one is identity — where that code does have to be very carefully scrutinized and needs extra review. So I would say, the first job is for architects to know where they need to be careful and where they can be a little more freelance. And the good news is, most places, you can kind of wing it and be okay. But there are definitely places like how you mount a volume, or how you do identity, or what [network interposition](#) you allow, where you have to be pretty darn careful. And then it’s good to have many people look at those things.”

The Identity People Know

Identity — specifically, the digital authentication of an agent in a system — is the focus of many Kubernetes security projects today, many of which are arguably (until otherwise noted) in scope. Since version 1.6 was introduced in March 2017, the orchestrator has supported role-based access control (RBAC), which in this instance is essentially a set of permissions that describe a role that may be attributed to an identity.

All that having been said, identity is not a native component of Kubernetes. There’s an architectural reason for this: Identity is one of those virtues that assumes a fixed state. In infrastructure security, identity is granted to things whose existence is presumably assured. By definition, a container is ephemeral. Its continued existence is assuredly limited.

What’s more, an identity is a state — a representation made by data of a particular property or characteristic.

“There’s a bunch of work in identity and secrets that we need to do,” said Microsoft’s Brendan Burns. “We really don’t have identity throughout the stack. We’re just starting to do things like add Active Directory integration for Azure Container Service; we’re just starting to think about, what does it mean to use a cloud-provided secrets service like [Key Vault](#) to provide secrets into your cluster?”

In an easier world to imagine, an application can identify itself to an authentication system by way of a digital certificate. In a world where the application is represented by a variable number of microservices, many of them redundant, scattered across the cloud, designed intentionally for statelessness, what would a single certificate be certifying?

And if you think about it further, a secret is, by definition, a state. Logically, it’s the very opposite of a stateless system; a secret is a unit of data that two components maintain independently from one another, to provide some sort of link between them. Imagine the session key shared between your browser and a web host. Now picture thousands of similar articles being shared simultaneously between microservices. If some services blink out of existence, how does the system know which remaining services are sharing what secrets?

“With secure pods, each pod would have its own identity that’s verifiable ... The correct way to think about it is, a single-tenant pod on a multi-tenant node.”

-Eric Brewer, Google.

If problems this fundamental are to be resolved “out of scope,” as Google’s Eric Brewer suggests, then the solution — wherever it comes from — will need to be integrated into Kubernetes very directly. It can’t be bolted on like an afterthought; it needs to be capable of addressing, and even augmenting, the basic architecture of the system. Otherwise, the link between the security package and the orchestrator will always be the weakest one.

“Kubernetes provides a number of different mechanisms that are very flexible,” explained Google’s Aparna Sinha. “There are different implementations that you can use on different clouds, or different on-premises environments. There are different mechanisms, obviously, for authentication, authorization and identity; but also for role-based access control, and previously, attribute-based access control. Then for securing the container interface — for adding security policies, whether that’s using [AppArmor](#), or [SELinux](#) or other Linux implementations.”

CoreOS has helped the CNCF assemble the [Kubernetes Security Release Process](#), which is a group of volunteer engineers who coordinate the response to vulnerabilities once they’re discovered and responsibly disclosed. It’s a way of automating the workflow that leads from patch discovery to remediation. But as the orchestrator’s architects have stated publicly, for Kubernetes to evolve to what can rationally be called a platform, it needs to adopt some fundamental identity tools, including means to reliably and exclusively identify nodes, pods and containers.

“Increasingly,” Sinha added, “we’re working on node security and pod-level security. The roadmap for Kubernetes to have security capabilities is fairly robust and extensive. Any particular implementation, on any particular cloud, will use these capabilities differently.”

The Extensibility People Expect

One of the reasons for the variation of capabilities across Kubernetes implementations is due to a recent, major alteration to its extensibility model.

“We had this old concept called [ThirdPartyResource](#),” explained Brewer. “This was one of the primary extension mechanisms that people were using — things like CoreOS operators, and other people doing things based on **ThirdPartyResource**, which lets you extend the Kubernetes API in sort of a native way.

“That trial by fire ... showed some weaknesses,” he admitted. “So that has evolved, and we’ve got a new replacement for it called the [CustomResourceDefinition](#) [CRD]. It’s a new iteration on that idea.”

Specifically, the CRD enables an outside service to identify a resource that the orchestrator will treat as though it were the same class as a pod (which is a native class of resource). So while we’ve come to think of Kubernetes as a maintainer of pods, or groups of containers, by virtue of this extensibility, it can equally instantiate and orchestrate other structures at a very low level.

““ At some level, security is a continuum. We want to make sure people can’t shoot themselves in the foot.”

-Brendan Burns, Microsoft.

“This gives a new opportunity for the ecosystem to offer third-party extensions to the Kubernetes system,” said Brewer, “that really, truly

behave like native things. This both gives us an opportunity to modularize, and vendors a chance to innovate and do creative things that we never would have thought about.”

If the final security model for the orchestrator truly has yet to be conceived, then CRD could throw open the floodgates for every conceivable idea of what that model should be. Typically, a security model for a platform must follow the existing context of that platform. CRD blows open the architectural model at the contextual level, letting outside parties go into the basic diagram, if you will, and draw rectangles with all new colors for completely ingenious purposes.

As Brewer further explained, Kubernetes had recognized the natural security boundary of a compute operation — the “compute boundary,” for short — as the virtual machine. “That’s the one that’s been well-tested and has a proven track record,” he said.

“Up until very recently, Kubernetes has made a cluster be a single entity, in terms of the security. If you have access to a cluster, you have access to all the nodes in the cluster. In [Kubernetes] 1.7 we’re making the node be a security boundary, separate from the other nodes in the cluster. That finer granularity is important for lots of different use cases. But it’s still based on the machine being the boundary.”

The Permutation No One Expected

Brewer said he’s personally in favor of the idea of the pod being a natural security boundary. One suggestion he’s already made to the Kubernetes community describes a secure pod — an extension of the basic pod class, whose behavior would differ substantially from that of the basic pod. Theoretically, he said, identity and authenticity of basic resources could be managed from within a secure pod, whose life cycle follows its own rules.

Pods could take on new permutations: for example, nested virtualization — literally pods within pods. Or perhaps a single pod might cohabit multiple VMs simultaneously. Why? In an environment whose architects extol the virtues of statelessness, identity — the ultimate instance of state in a system — may be pretty difficult to reconcile. And even though Kubernetes makes good attempts at trying to reconcile the issue of statefulness by means of etcd — which one CoreOS developer calls “the keeper of the state” — identity is not something representable by a key/value store.

“In my mind, this is the real cloud that I’ve wanted for a long time — which is not based on moving your machine from a data center on-prem to a data center in the cloud.”

-Eric Brewer, Google.

Accomplishing this quasi-dimensional model, where state and statelessness can co-exist, requires a decision on the part of the Kubernetes community as to the identity and location of its new security boundary — the entity to which policy refers. Already, said Eric Brewer, that boundary has moved from the cluster to the node, though in the future he would like for it to be even finer-grained. “That’s something I have to work with the community to establish,” he said.

“With secure pods, each pod would have its own identity that’s verifiable, and it would not be able to interfere with the other pods on the same physical machine ... The correct way to think about it is, a single-tenant pod on a multi-tenant node.”

The Accessibility People Demand

Verifiability in this context mandates that Kubernetes' engineers not re-invent the wheel here. The secure part of the pod will need to be recognizable to whatever system presently identifies entities in existing data centers. For this reason, the orchestrator's engineers will need to adapt the platform further, to embrace network policies that already exist, in whatever forms they exist.

"I think what happened with [the Network Policy API](#) is a good model going forward," said Gabe Monroy, Microsoft Azure's principal program manager for containers, and the former chief technology officer of Deis — the producer of the Helm deployment manager for Kubernetes, recently acquired by Microsoft.

“Instead of everything running in one monolithic application for your supply chain, the things that make up the service you deliver come from a bunch of different pieces and parts. And that supply chain is APIs.”

-Brian Gracely, Red Hat.

"By default, prior to Kubernetes 1.3, network access control and ingress/egress policies inside the cluster were basically wide open," Monroy continued. "And a number of security vendors with a networking focus got together and said, 'Look, we need to fix this, but we need to do it in an interoperable way.' The result of that work was the Kubernetes Network Policy API, which specifies basically a schema and a way to do default-

deny access policy across a cluster; and then a way to use the Kubernetes label selectors to define which containers can talk to each other, both for ingress and egress.”

The model Monroy refers to that he likes is not so much the policy model as the collaboration that made it feasible. The participation of a plurality of security vendors, he said, enabled them to come up with a system that was not a least-common denominator for policy, but rather a platform with decoupled and interchangeable components.

One project whose goal is to collect vulnerability data from containers, and put that data to use in a protection mechanism, is [CoreOS' Clair](#). CoreOS is a participant in Kubernetes' Network Policy API. Another is Twistlock, whose principal product is a commercial container security platform.

As Twistlock CTO [John Morello](#) remarked, it won't help companies in his position for any orchestrator's security architecture to be too specific to itself, “such that we build separate things for Kubernetes versus [Docker] Swarm versus [Mesosphere] DC/OS. The more standardization there is, in terms of the ways an orchestrator might say, ‘Before I run something, there's a standard way that I'm going to check to see with some external service whether I should run this ...’ that would make it easier for customers to integrate those tools together.”

Already, Twistlock customers can deploy its security console as a Kubernetes pod, and deploy its Defender agents to multiple nodes simultaneously by way of [a specified DaemonSet](#) — a policy that mandates the orchestrator run specified pods within each node. But the methodologies for accomplishing this today may be a bit too Kubernetes-specific for enterprises to adopt them in large numbers. By instituting a more abstract, multi-level, decoupling mechanism, Kubernetes developers may ensure that whatever policies run outside the platform

are, by definition, never fine-tuned solely for Kubernetes.

“Each of those security vendors do provide a different policy enforcement backend for those things,” stated Microsoft’s Gabe Monroy. “I think that’s a good model going forward; that said, there are cases where people are going to have unique value that they want to add, and they may want to do it outside the context of the Kubernetes community. I think we should welcome that, too, because there are types of innovation that are going to warrant both approaches.”

What Will the Platform Be in Three Years’ Time?

By naming this volume “The State of the Kubernetes Ecosystem,” we make a clear presumption that the ecosystem — the establishment of cyclical economic processes intended to benefit all of their participants — revolves around Kubernetes, and will continue to do so. Yet this is actually an open question.

“For a long time, when people were very intimidated by the complexity of Kubernetes, I always encouraged them to think about where Kubernetes came from, why it exists, and what kinds of problems it was trying to solve.”

-Laura Frank, Codeship.

In asking DevOps professionals for the reasons they believed Linux assumed a pinnacle position in their data centers — often ousting Windows Server from its perch — many will tell you that Linux enabled the creation of platforms that would never have happened before, and that those platforms in turn cemented Linux' place among servers and in the cloud. At a certain point in history, Linux was a passenger on this journey.

Some believe the history of Docker will take similar turns. And many see such a path for Kubernetes.

“I believe, fundamentally, that Kubernetes is just a small part of this cloud-native revolution,” remarked Google's Tim Hockin, “and hopefully it's a catalyst for the way people think about how to run their workloads. But just like Linux, it is not the end goal in and of itself. Now, three years isn't that long of a time horizon, but I think by three years from now, we'll be well down this path towards the cloud-nativization efforts, that we will hopefully see a lot more enterprise adoption of containers and Kubernetes. These things are doing great, but enterprise is very slow to take off.”

Hockin would like to see, by 2020, the command-line tools that have come to define Kubernetes in its short lifespan, such as **kubect1**, become the engines of broader, client-side toolsets — for example, like [ksonnet](#), an implementation of Google's JSON templating language called **jsonnet**, that compiles higher-level declarative code into Kubernetes-native YAML files. The richer the abstractions available to developers and operators on the client side, he believes, the more likely that organizations will find themselves adopting Kubernetes without even really knowing they've done so.

“The roadmap for Kubernetes to have security capabilities is fairly robust and extensive. Any particular implementation, on any particular cloud, will use these capabilities differently.”

-Aparna Sinha, Google.

“I hope that, within three years, Kubernetes is just sort of assumed to exist,” stated Hockin, “and it becomes just a tool in the toolbox like a Linux server. You remember, there was once a time when the idea of bringing up a Linux server was something you had to really waffle over, debate, and talk to your boss about. And now, when you talk about servers, Linux is just sort of assumed as the default. And when people tell me they’re bringing up a Windows Server, that’s the exception, not the rule.”

Red Hat’s Brian Gracely believes the Kubernetes brand will not be so sublimated by 2020.

“I absolutely think we will have people talking about Kubernetes. It’s still only a couple of years old,” said Gracely. “We look at other projects that have lasted for periods of time, and there are still communities around the technology. I think the team that works on OpenShift is going to be very focused on, how are we helping customers become really successful in a fast period of time, to make a change to my business? We’re going to spend the next couple of years getting platforms deployed all over the place, people realizing the value of not having to build your own platform.

And then I think we'll see customers who say, 'Look, those first few flowers began to bloom. How do I scale this?'"

"I think in three years, we'll be talking about all the things that people are building with things that run on top of Kubernetes," pronounced CoreOS' Caleb Miles. "I believe we'll see amazing [TensorFlow](#) operators, open source machine learning tools, the next-generation data stores, and traditional relational databases. I think where we'll be in three years is talking about how easy it is for new entrants to build and deploy complex applications, on top of modern platforms like Kubernetes."

But the attention that these applications and use cases would be generating, may come at the expense of the platform supporting it all.

"I believe that certainly the details of lower levels of the infrastructure will begin to fade," Miles continued. "In 2017, we're talking a lot about the applications we're building that have moved beyond a single node — a single server node, desktop node or a node in the data center ... I think we will continue to see that trend: building more and more complex applications, and deploying them to platforms. And I think that conversation will be about the things that we're building, and less about the platform itself. People find value in the platform, in order to deliver value to their users — in the same way that Linux is ubiquitous, but not much talked about at the distribution level anymore."

[Ihor Dvoretzkyi](#), who manages the Kubernetes product line for commercial platform maker Mirantis, agrees ... up until the part where it drops out of the public conversation. "I would compare Kubernetes with Linux, and I would name Kubernetes as Linux for cloud-native applications," Dvoretzkyi said.

"I can see the ecosystem of Kubernetes will grow, but at the same time, the Kubernetes core will focus mostly on stability," he continued. "I can

see many vendors, who are producing Kubernetes as the core for their commercial projects, actively using it right now. The same situation with end users: Many companies that have switched from legacy technologies to cloud-native to microservices, are using Kubernetes for their new solutions. And I'm really happy to see that. And I'd like to see that in three years, but with a much bigger scale in the market."

“There are definitely places like how you mount a volume, or how you do identity, or what network interposition you allow, where you have to be pretty darn careful.”

-Eric Brewer, Google.

“I think that conformance, stability, and project and community health are critical to the future that we're imagining,” said Google's Aparna Sinha.

“But assuming that we get those things right, I think what will happen with Kubernetes is that it will become increasingly the container orchestration platform that is used across different infrastructure environments. And it will become the most deployed, de facto standard for that — assuming we get the other three things right.

“Right now, there are a lot of different distributions of Kubernetes,” Sinha continued. “We think there will be some that will have critical mass, and users will use a set of distributions on multiple, different clouds, in multiple, different environments, and be able to make their workloads portable across those environments. The second piece of it is, what's on top of Kubernetes. And I think that there's a host of different developer tools that will be developed, both within [Google Cloud Platform] as well

as in the community. Those will make it such that developers do not need to think about Kubernetes itself, and will be able to run their applications and achieve the benefits — for example, integration with CI/CD systems, logging, monitoring and all the other aspects that developers rely on. Those things that there are many of today — as the technology matures, there will be a few that will emerge as the best tools, and those will become used more and more.”

“If we’re going to survive as a project, we have to do this: We have to allow for flexibility and heterogeneity and openness,” said Microsoft’s Brendan Burns, “and people building solutions on top and extending the project outside of the core project, in order for us to not become stuck in what we have done before, and enable the breadth of community that we need to, in order to truly be successful.”

Burns foresees the Kubernetes ecosystem evolving into an ecosystem where it’s one of many important components.

“A programming language has its core programming syntax, and then it has maybe a standard library that the language owners are also responsible for, but usually a broader group of people,” Burns continued. “And then there’s an ecosystem of libraries that develop around it. And I think Kubernetes will be the same way. We want to have a really crisp, well-defined, principled core, maybe some standard patterns, and then a really rich ecosystem of libraries, utilities, and tools that people can mix-and-match to find the solution that works for them. That’s the ideal way that you survive. Otherwise you just become a niche product that only works in a very specific set of circumstances.”

“Kubernetes,” believes Google’s Eric Brewer, “is going to be the platform of platforms.

“As for any domain that’s specialized ... you can build any kind of Platform

as a Service using Kubernetes as your substrate. And that is very powerful. In fact, it solves one of the problems with domain-specific platforms: If it's too domain-specific, you get some leverage from that, but you also can get stuck there, and it's hard to get out and do things the platform wasn't intended for. If you build something domain-specific on top of Kubernetes, and you want a little more than the domain offers, you can back off to Kubernetes. And now you have a very flexible solution to a wide range of problems.

“But also, taking a step back,” Brewer continued, “in my mind, this is the real cloud that I've wanted for quite a long time — which is not based on moving your machine from a data center on-prem to a data center in the cloud. That is a useful transition. But this transition is much more useful. This is about, let's program to processes and services and APIs, and make that our foundation for building the cloud. That's a much more powerful transformation than moving VMs into someone else's data center.”

Every adoption and implementation of a digital technology platform has been a process of transition. What's different with Kubernetes is that it's about thinking smaller to get bigger. The transition to microservices mandates a mindset shift to a very foreign way of thinking about business processes — both the ones we're transitioning and the new ones we're creating. But we could very well find ourselves at the tail end of the transition with less junk left over — less virtual machine overhead, reduced latency, far greater performance. This time, we could be automating the things worth automating.

CLOSING

Container orchestration is a genuinely new concept in the history of computing. It is the modern manifestation of web services at the most granular of levels. In the late 1990s, when the concept of web services was created, software developers and network engineers toyed with the idea of building directories of common functions that could be called remotely through common interfaces. For a time, the tech press concocted a war between Microsoft and Novell for the right to set the standard for such interfaces.

When knowledge management applications first tested the limits of ordinary servers, a new type of load balancing scheme was created — one that listened to the type and context of the requests being received, and that responded by dispatching the call to a server based on its availability. When that server became virtual, the dispatch call became easier, and the network supporting that dispatch could become software-defined.

But now, the server in that model has become an individual function: a microservice. And now, Kubernetes and the ecosystem that incorporates it have made it feasible for a very granular service, anywhere in the network, to respond to a remote call from anywhere on the planet. The dispatching mechanism, having manifested itself from those first web applications, has become a sophisticated proxy. And the means for maintaining these individual services are being hardened and resolved into pipelines. Now, these services can be serviced and maintained on an individual basis, and the process of overseeing the entire scope of the deployed application life cycle can be automated.

“Kubernetes Deployment Patterns and Pipelines” will be the second book in our series about the Kubernetes ecosystem. It will focus on the

CLOSING

mechanisms that support this new class of applications and the infrastructure on which they're built. Until then, take good care of you and yours, and we'll see you on [The New Stack](#).

KUBERNETES SOLUTIONS DIRECTORY

Although this directory has almost a hundred entries, it is not meant to be comprehensive. Instead, it lists many of the projects and vendor offerings that are used to deploy and manage Kubernetes and the applications running on it. Listings are divided into four sections to make it easier for the reader to quickly review, and are only a starting point when reviewing solutions you may want to use or consider.

KUBERNETES DISTRIBUTIONS

Platforms, products, services and projects that include Kubernetes as a distribution.

Product/Project (Company or Supporting Org.)	Type of Distribution
APPUiO (APPUiO) Platform as a Service (PaaS) based on Red Hat OpenShift. The Swiss company targets developers as customers.	Platform
Canonical Distribution of Kubernetes (Canonical) Canonical's distribution provides customers access to stable upstream Kubernetes releases, as well as access to early builds of the upstream Kubernetes development branch. Canonical has optimized Kubernetes to run with its existing infrastructure and DevOps tools, but it also works across all major public clouds and private infrastructure.	Platform
CloudStack Container Service (ShapeBlue) A Container as a Service (CaaS) solution that combines the power of Apache CloudStack and Kubernetes. It uses Kubernetes to provide the underlying platform for automating deployment, scaling and operation of application containers.	Platform
Container Platform (Greenqloud) Cloud management system with Kubernetes container and cluster engine built in.	Platform
Deis Workflow (Microsoft) A Kubernetes-native PaaS focused on developer self-service and operational flexibility. Deis Workflow helps teams quickly get up and running with Kubernetes on any public cloud, private cloud or bare metal cluster.	Platform
Diamanti (Diamanti) A purpose-built container infrastructure that addresses the challenges of deploying containers to production while letting users keep their existing infrastructure. It does switching native on bare metal, plugging into a CPU bus.	Platform
fabric8 (Red Hat) Applications can read and write data into etcd. A simple use case is to store database connection details or feature flags in etcd as key-value pairs. These values can be watched, allowing your app to reconfigure itself when they change.	Platform
Fission (Platform9) A framework for serverless functions on Kubernetes.	Platform
FusionStage (Huawei) An enterprise-grade Platform as a Service product, the core of which is based on mainstream open source container technology including Kubernetes and Docker. It is available for both public cloud and private data center deployment.	Platform

Product/Project (Company or Supporting Org.)	Type of Distribution
Getup Cloud (Getup Cloud) A platform built with Docker, Kubernetes and OpenShift. It is currently offered as a trial by a Brazilian startup.	Platform
Giant Swarm (Giant Swarm) A hosted container solution to build, deploy and manage containerized services with Kubernetes as a core component. It offers customers fully-managed private Kubernetes clusters, including management of master and nodes. It is offered "as a service" or can be deployed and managed on premises by Giant Swarm.	Platform
Google Container Engine (Google) Google Container Engine is a cluster management and orchestration system that lets users run containers on the Google Cloud Platform.	Platform
Hasura Platform (34 Cross Systems) A platform for creating and deploying microservices. This emerging company's infrastructure is built using Docker and Kubernetes.	Platform
Hypernetes (HyperHQ) A multi-tenant Kubernetes distribution. It combines the orchestration power of Kubernetes and the runtime isolation of Hyper to build a secure multitenant container management platform.	Platform
IBM Bluemix Container Service (IBM) Use IBM Containers to run Docker containers in a hosted cloud environment on IBM Bluemix. IBM Containers provide full hosting and life cycle management of Docker containers, along with automatic and integrated log analytics and monitoring, elastic scaling with auto-recovery, reliability tools, load balancing and routing, persistent storage, security services and a private image registry.	Platform
K2 (Kraken 2) (Samsung CNCT) Enables deployment of a Kubernetes cluster using Terraform and Ansible on top of CoreOS.	Platform
Kel (Eldarion) An open source, Kubernetes-based PaaS built in Python and Go, that makes it easy to manage web application deployment and hosting through the entire software life cycle.	Platform
Kismatic Enterprise Toolkit (KET) (Apprenda) KET is Apprenda's commercially supported and fully open source Kubernetes offering. It provides a set of default cluster services that go beyond the basic automation of installing and running Kubernetes on a few nodes or on a laptop.	Platform
Kontena (Kontena) Kontena is a container orchestration tool. It abstracts containers into application services and establishes an internal network between linked services, making it easy to deploy and scale applications across multiple hosts.	Platform
Kops (Cloud Native Computing Foundation) Kubernetes Operations (kops) enables a production-grade Kubernetes installation, upgrades and management.	Platform
KUBE2GO (Platform9) Tool to deploy Kubernetes clusters to public clouds. As of publication, only AWS is supported.	Platform

Product/Project (Company or Supporting Org.)	Type of Distribution
Kubernetes (Cloud Native Computing Foundation) Kubernetes is an open source Docker orchestration tool. Google initially developed Kubernetes to help manage its own LXC containers. Stateful support is done through a new object called Pet Set. In addition, there are many networking and data-volume plugins available.	Platform
Kubernetes Services Managed by LiveWyer (LiveWyer) The consulting company manages Kubernetes implementations and provides Kubernetes training.	Platform
Kubo (Pivotal) Provides a solution for deploying and managing Kubernetes with BOSH alongside Cloud Foundry.	Vendor
Last.Backend (Last.Backend) A platform built on top of Kubernetes with a command-line toolkit and UI to deploy apps and manage infrastructure.	Platform
Managed Kubernetes (Kumina) Dutch consulting company that provides managed services.	Vendor
Minikube (Cloud Native Computing Foundation) Minikube is a tool that makes it easy to run Kubernetes locally. Minikube runs a single-node Kubernetes cluster inside a virtual machine (VM) on your laptop. It is for users looking to try out Kubernetes or develop with it day-to-day.	
OpenShift Container Platform (Red Hat) A container application platform that can span across multiple infrastructure footprints. It is built using Docker and Kubernetes technology.	Platform
OpenShift Origin (Red Hat) OpenShift Origin is the upstream open source version of OpenShift and is meant to allow for development of cloud-native applications. OpenShift is a PaaS built on Docker containers that orchestrates with Kubernetes. It also has Atomic and Red Hat Linux components.	Platform
Photon Platform (VMware) A container-optimized cloud platform that provides on-demand access to Kubernetes clusters.	Vendor
Pivotal Container Service (Pivotal) A commercial version of Kubo that makes it easy to deploy Kubernetes and consume it in environments running vSphere or Google Cloud Platform.	Vendor
Platform9 Managed Kubernetes (Platform9) Kubernetes offered as a managed service. Customers can utilize Platform9's single pane of glass, allowing users to orchestrate and manage containers alongside virtual machines (VMs). In other words, you can orchestrate VMs using OpenStack and/or Kubernetes.	Vendor
Rancher (Rancher Labs) Rancher natively supports and manages Kubernetes, Mesos and Swarm clusters.	Vendor

Product/Project (Company or Supporting Org.)	Type of Distribution
Red Hat OpenShift (Red Hat) Integrated, web-based developer environment based on the Eclipse Che project (acquisition of Codenvy), source code repository and CI/CD pipeline. Development environment integrated with OpenShift Online.	Platform
Red Hat OpenShift Container Platform (Red Hat) A container application platform that can span across multiple infrastructure footprints (bare metal, virtual machine, VMware, OpenStack, AWS, Azure and GCP). It is built using Docker and Kubernetes technology. It integrates multitenant networking (SDN), multiple types of storage, container registry, Red Hat middleware and application services, and Open Service Broker. It runs on RHEL hosts, is deployed using Ansible and managed with CloudForms.	Platform
OpenShift Dedicated (Red Hat) Private, high availability OpenShift cluster, hosted on Amazon Web Services (AWS) or Google Cloud Platform, and operated as a cloud service by Red Hat.	Platform
Red Hat OpenShift Online (Red Hat) Red Hat's public cloud version of OpenShift that developers around the world can consume as a service (free and paid tiers). It is built using Docker and Kubernetes technologies.	Platform
StackPointCloud (StackPointCloud) Allows users to easily create, scale and manage Kubernetes clusters of any size with the cloud provider of their choice. Its goal is to be a universal control plane for Kubernetes clouds.	Vendor
Supergiant (Qbox) Supergiant is an open source framework that runs Docker containers. It hosts stateful, clustered applications utilizing Kubernetes under the hood. It uses its own structures and code for persistent storage and external load balancing. Qbox, the creator of Supergiant, provides commercial support.	Vendor
SUSE Container as a Service Platform (SUSE) An application development and hosting platform for container-based applications and services. It uses SUSE Linux Enterprise MicroOS and Kubernetes.	Vendor
Symphony (Stratoscale) Managed Kubernetes offered as a service.	Vendor
Tectonic (CoreOS) Tectonic is the enterprise-ready Kubernetes solution that delivers pure, upstream Kubernetes. Tectonic provides automated operations allowing users to easily upgrade to the latest Kubernetes software version with one click, enables portability across private and public cloud providers, and is always secure with LDAP, RBAC and SAML support. It is secure and simple so organizations can easily scale applications, deploy consistently, and easily manage applications across environments. Along with the most current release of Kubernetes, Tectonic also includes installers to help get you up and running quickly, a console to visually investigate your cluster, operators to manage your cluster components, and security features to allow you to integrate with your existing security frameworks.	Vendor

Product/Project (Company or Supporting Org.)	Type of Distribution
Telekube (Gravitational) A toolkit for packaging, deploying and remotely managing complex multi-node Linux applications across clouds and on premises, all over the world. It bills itself as a private SaaS platform.	Vendor
TenxCloud Container Engine (TCE) (TenxCloud) A Kubernetes service offered by a Chinese company.	Vendor

TOOLS AND SERVICES

Offerings that help with the implementation of Kubernetes, as well as the deployment and management of applications on top of Kubernetes.

Product/Project (Company or Supporting Org.)	Type of Professional Service (if applicable)
AppController (Mirantis)	
A pod that can be deployed to a Kubernetes cluster to create objects and manage dependencies.	
Ark (Heptio)	
A utility for managing disaster recovery, specifically for Kubernetes cluster resources and persistent volumes.	
Azure Container Service (Microsoft)	
Azure Container Service simplifies the creation and configuration of a cluster. The default configuration of this cluster includes Docker and Docker Swarm for code portability; and Marathon, Chronos and Apache Mesos to ensure scalability.	
Bootkube (N/A)	
A helper tool for launching self-hosted Kubernetes clusters.	
Cabin (Bitnami)	
An iOS and Android application for managing Kubernetes applications.	
cAdvisor (N/A)	
cAdvisor (Container Advisor) is a Google-supported project that analyzes resource usage and performance characteristics of running containers.	
Containerd (Cloud Native Computing Foundation)	
A daemon to manage containers on one machine. It is based on the Docker Engine's core container runtime and follows Open Container Initiative specifications.	
ContainerPilot (Joyent)	
Works in conjunction with other schedulers — letting them start and stop containers — with ContainerPilot orchestrating the rest. Applications orchestrated by ContainerPilot are portable from one scheduler to another.	
Datadog-Kubernetes Integration (Datadog)	
Collects and monitors metrics from Kubelets in real time. It is deployed as a Docker container alongside existing workloads.	

Product/Project (Company or Supporting Org.)	Type of Professional Service (if applicable)
Digital Rebar (RackN)	
A container-ready cloud and hardware-provisioning platform.	
ElasticKube (CenturyLink)	
A service for connecting CI/CD pipelines, configuration management tools, and deploying cloud applications. It is an an open source management platform for Kubernetes that promotes self-service for containerized applications.	
Endocode (Endocode)	Consulting
A German software engineering firm that has helped provided many contributions to several container-related projects. Provides consulting services associated with Kubernetes.	
Heapster (Heapster)	
Enables analysis of compute resource usage and monitoring of container clusters. Heapster currently supports Kubernetes and CoreOS natively.	
Helm (Cloud Native Computing Foundation)	
A Kubernetes-native package manager that helps operators declare and manage complex, multi-part applications.	
Heptio Professional Services and Support (Heptio)	Support
Heptio is a company by founders of the Kubernetes project, built to support and advance the open Kubernetes ecosystem.	
Jetstack Container Services (Jetstack)	Support
Jetstack is a consulting company focused on helping companies build a container management infrastructure.	
K8S Dashboard (Distelli)	
Distelli provides a dashboard to dieplay and manage applications.	
K8sPort (Cloud Native Computing Foundation)	
A social network with gamification features that supports the Kubernetes community.	
Kolla-Kubernetes (OpenStack Foundation)	
The project provides Docker containers and Ansible playbooks to deploy Kubernetes on OpenStack.	
Kompose (Cloud Native Computing Foundation)	
A tool to help users familiar with docker-compose move to Kubernetes.	
ksonnet (Heptio)	
Jsonnet is an open source JSON templating language from Google. ksonnet-lib and kubecfg provide a simpler alternative to writing complex YAML for Kubernetes configurations.	
kubeadm (Cloud Native Computing Foundation)	
A part of the Kubernetes distribution that helps install and set up a Kubernetes cluster.	

Product/Project (Company or Supporting Org.)	Type of Professional Service (if applicable)
Kubediff (Weaveworks)	
A tool for Kubernetes to show differences between running state and version-controlled configuration.	
Kubeflix (Red Hat)	
Provides Kubernetes integration with Netflix open-source components such as Hystrix, Turbine and Ribbon.	
Kubeless (Bitnami)	
A Kubernetes native serverless framework. It supports both HTTP and event-based triggers, has a serverless plugin, a graphical user interface and multiple runtimes.	
Kubermatic (Loodse)	
Makes it easy to deploy and manage multiple container clusters.	
Kubernauts (Kubernauts)	
Organized as a non-profit, Kubernauts provides training and consulting services. It manages the Kubernauts Worldwide Meetup.	
Kubernetes Anywhere (Kubernetes Anywhere)	
An automated solution that will eventually allow users to deploy Kubernetes clusters across multiple clouds.	
Kubernetes Dashboard (Kubernetes Dashboard)	
A general purpose, web-based UI for Kubernetes clusters. It allows users to manage applications running in the cluster and troubleshoot them, as well as manage the cluster itself.	
Kubernetes service-catalog (N/A)	
Works with the Open Service Broker API to integrate service brokers with Kubernetes. It provides a way for Kubernetes users to consume services from brokers and easily configure their applications to use those services.	
Kubernetes Support (Apprenda)	
Professional support for Kubernetes to handle both original implementation and ongoing operations. Apprenda offers three tiers of support, including pay per incident.	
KubernetiC (Harbur Cloud Solutions S.L.)	
A desktop client to manage Kubernetes clusters.	
Kublr (EastBanc Technologies)	
An automated cluster management platform.	
Kubespray (N/A)	
A tool to deploy Kubernetes clusters. It is an alternative to kops and kubeadm.	
Magnum (OpenStack Foundation)	
An OpenStack API service which makes container orchestration engines, such as Docker and Kubernetes, available as first class resources in OpenStack.	

Product/Project (Company or Supporting Org.)	Type of Professional Service (if applicable)
Open Service Broker API (Cloud Foundry Foundation)	
The project gives developers, ISVs and SaaS vendors a way to deliver services to applications running within cloud-native platforms such as Cloud Foundry, OpenShift, and Kubernetes. It works with the service-catalog project that is in the Kubernetes incubator.	
Poseidon (University of Cambridge)	
Poseidon is Firmament's integration with Kubernetes.	
Prometheus (Cloud Native Computing Foundation)	
Prometheus is an open source systems monitoring and alerting toolkit, service monitoring system and time series database.	
Quick Start for Kubernetes (Heptio)	
A set of templates and configurations to quickly set up a Kubernetes cluster on AWS using CloudFormation and kubeadm.	
ReactiveOps (ReactiveOps)	Consulting
Custom builds DevOps platforms based on Kubernetes.	
rkt (Cloud Native Computing Foundation)	
rkt is a command-line interface (CLI) for running app containers on Linux based on the App Container Specification (appc spec).	
Sematext Kubernetes Agent (Sematext)	
Provides operational insights by collecting Kubernetes logs, events and metrics with out-of-the-box metrics charts, searchable logs, and the ability to correlate logs, metrics, alerts and more. It utilizes Sematext Docker Agent to extract information from Docker container names, and tags all logs with name space, pod, container, image name and UID.	
Sonobuoy (Heptio)	
A diagnostic tool that makes it easier to understand the state of a Kubernetes cluster by running a set of Kubernetes conformance tests in an accessible and non-destructive manner.	
Supergiant Support (Supergiant)	Support
Supergiant is an open source framework that runs Docker containers. It hosts stateful, clustered applications utilizing Kubernetes under the hood. It uses its own structures and code for persistent storage and external load balancing. Qbox, the creator of Supergiant, provides commercial support.	
Tack (N/A)	
An alternative to using CloudFormation. It is an opinionated Terraform module for creating a highly available Kubernetes cluster running on Container Linux in an AWS Virtual Private Cloud.	
Virtual Private Pipelines (Oracle)	
A Docker native, single tenant and fully managed CI/CD platform optimized for Kubernetes and working with microservices. It offers network isolation and flexible concurrency.	
Weave Scope (Weaveworks)	
Weave Scope offers a real-time monitoring solution for containers.	

RELEVANT DEVOPS TECHNOLOGIES

Tools and technologies that work with Kubernetes throughout the DevOps life cycle. Entries are defined as primarily helping with the create, package/release, configure or monitoring steps.

Product/Project (Company or Supporting Org.)	DevOps Life Cycle Segment
AppDynamics (Cisco) Application and business performance software that collects data from agents installed on the host. It provides an extension to collect data from the Docker API.	Monitoring
AppFormix (Juniper Networks) Cloud infrastructure monitoring and analysis software that runs in any public, private, multi-tenant or hybrid environment. It includes ContainerFlow, which utilizes Intel Resource Director technology to enforce isolation between workloads and deliver better performance for container workloads. The company is focused on analytics for operators of OpenStack and Kubernetes.	Monitoring
AppsCode (AppsCode) Integrated platform for collaborative coding, testing and deployment of containerized apps. Support is provided for deploying containers to AWS and Google Cloud Platform.	Create
Clocker (Cloudsoft) Clocker creates and manages Docker cloud infrastructures. It contains Apache Brooklyn blueprints to enable deployment and management of Docker Swarms and Kubernetes clusters.	Configure
CloudPlex (CloudPlex) A cloud orchestration and management platform. It uses Chef to deploy to VMs, and Kubernetes to deploy to Docker containers.	Package/Release
Cobe.io (Cobe.io) Provides a live topology of heterogeneous infrastructure, on top of which model performance metrics and alerts are overlaid.	Monitoring
Codeship Pro (Codeship) Codeship Pro is a fully customizable continuous integration platform with native Docker support in the cloud. It makes it easy to test and deploy your microservices and push to any registry. It's also perfect if you want to deploy with Kubernetes, as it comes with a convenient local CLI tool that allows you to run your builds locally, helps encrypt environment variables, and guarantees 100% parity between your development and production environment. Codeship Pro comes with a free plan that grants 100 builds per month, with unlimited projects, teams and users.	Package/Release

Product/Project (Company or Supporting Org.)	DevOps Life Cycle Segment
Container Builder (Google) Fast, consistent, reliable builds on Google Cloud Platform.	Create
Container Linux (CoreOS) CoreOS Container Linux is a minimal operating system that supports popular container systems out of the box. The operating system is designed to be operated in clusters. For example, it is engineered to be easy to boot via PXE and on most cloud providers.	Create
Draft (Microsoft) A tool for developers to create cloud-native applications on Kubernetes. Draft is still experimental.	Package/Release
Dynatrace (Dynatrace) Dynatrace's new suite of monitoring tools is based on its Ruxit technology. Its agent is injected into a container, which then autodiscovers new services running on a host and can fetch data from the Docker API. Dynatrace is also developing artificial intelligence technology to help with root cause analysis.	Monitoring
etcd (CoreOS) etcd is a distributed key-value store that provides a reliable way to store data across a cluster of machines. It's open source and available on GitHub, and is the primary datastore for Kubernetes. etcd gracefully handles leader elections during network partitions and will tolerate machine failure, including the leader. Your applications can read and write data into etcd. A simple use case is to store database connection details or feature flags in etcd as key-value pairs. These values can be watched, allowing your app to reconfigure itself when they change.	Configure
Fluentd (Cloud Native Computing Foundation) Fluentd is an open source data collector for unified logging layers.	Monitoring
Forge (Datawire) Builds services based on Docker and Kubernetes. YAML files are used to specify deployment configurations.	Configure
gRPC (Cloud Native Computing Foundation) A high performance, open source, general remote procedure call (RPC) framework that puts mobile and HTTP/2 first.	Support
Istio (N/A) A platform to integrate microservices, manage traffic flow across microservices, enforce policies and aggregate telemetry data. Istio's control plane provides an abstraction layer over the underlying cluster management platform.	Monitoring
Kong (Mashape) Kong is a management layer for APIs. It has the capability of orchestrating Dockerfiles.	Configure
Linkerd (Cloud Native Computing Foundation) An out-of-process network stack for microservices. It functions as a transparent RPC proxy, handling everything needed to make inter-service RPC safe, including load-balancing, service discovery, instrumentation and routing. Linkerd is built on top of Finagle.	Monitor

Product/Project (Company or Supporting Org.)	DevOps Life Cycle Segment
Loom (Datawire) Self-service provisioning for microservices in Kubernetes running on AWS. It has pre-configured models for creating development Kubernetes clusters in AWS.	Configure
Navigator (Jetstack) Managed DBaaS on Kubernetes, Navigator is a centralized controller for managing common stateful services on Kubernetes.	Configure
New Relic APM (New Relic) Application performance monitoring is at the heart of New Relic's suite of products, which it starting to call Digital Intelligence Platform. Its agent-based approach is particularly good for troubleshooting code-related application performance issues.	Monitoring
OpenTracing API (Cloud Native Computing Foundation) Consistent, expressive, vendor-neutral APIs for distributed tracing and context propagation.	Monitoring
Project Atomic (Red Hat) Project Atomic hosts run applications in Docker containers with components based on RHEL, Fedora and CentOS. In addition to Atomic Host, the project includes Nulecule, a container-based application specification that enables the use of existing containers as building blocks for new applications.	Create
Puppet Module for Kubernetes (Puppet) A templated configuration file to deploy Kubernetes with Puppet.	Configure
Red Hat OpenShift Application Runtimes (RHOAR) (Red Hat) Currently in beta, RHOAR is a set of cloud-native, container-optimized application runtimes based on Spring Boot, Eclipse Vert.x, Node.js and WildFly Swarm. Natively integrated with OpenShift Container Platform and Kubernetes.	Create
StackState (StackState) A full stack monitoring solution that provides container monitoring.	Monitoring
Sysdig Cloud (Sysdig) Based on open source Sysdig technology, Sysdig Cloud monitors, troubleshoots and alerts on containerized environments. Sysdig Cloud can be used as a cloud service or deployed as hosted software in your private cloud.	Monitoring
Tack (N/A) A Terraform module for creating Kubernetes clusters running on Container Linux by CoreOS in an AWS virtual private cloud.	Package/Release
Telepresence (Datawire) Enables local development against a remote Kubernetes or OpenShift cluster.	Create
Terraform (HashiCorp) Terraform is a tool to build and launch infrastructure, including containers.	Configure

Product/Project (Company or Supporting Org.)	DevOps Life Cycle Segment
Wavefront (VMware) Uses cAdvisor to collect container metrics, which are analyzed along with metrics from other systems and applications.	Monitoring
Weave Cloud (Weaveworks) SaaS that simplifies deployment, monitoring and management for containers and microservices. It integrates with Kubernetes and provides Prometheus monitoring as a service.	Monitoring

RELEVANT INFRASTRUCTURE TECHNOLOGIES

The following include common examples of the storage, networking, compute and other infrastructure technologies that enable the use of cloud-native environments like Kubernetes.

Product/Project (Company or Supporting Org.)
<p>CoreOS Container Linux (CoreOS)</p> <p>CoreOS Container Linux is a minimal operating system that supports popular container systems out of the box. The operating system is designed to be operated in clusters. For example, it is engineered to be easy to boot via PXE and on most cloud providers.</p>
<p>Container Network Interface (CNI) (Cloud Native Computing Foundation)</p> <p>CNI is a project to help configure network interfaces for Linux application containers. It helps set up network connectivity of containers and remove allocated resources when the container is deleted.</p>
<p>CNI-Genie (Huawei)</p> <p>Enables container orchestrators to seamlessly connect to choice of CNI plugins like Calico, Canal, Romana and Weave.</p>
<p>Container Registry (Google)</p> <p>Fast, private Docker image storage on Google Cloud Platform.</p>
<p>Contiv (Cisco)</p> <p>Unifies containers, VMs, and bare metal with a single networking fabric, allowing container networks to be addressable from VM and bare-metal networks.</p>
<p>dex (CoreOS)</p> <p>dex is an identity service that uses OpenID Connect to drive authentication for other apps. Dex runs natively on top of any Kubernetes cluster. dex is not a user-management system, but acts as a portal to other identity providers through “connectors.” This lets dex defer authentication to LDAP servers, SAML providers, or established identity providers like GitHub, Google and Active Directory. Clients write their authentication logic once to talk to dex, then dex handles the protocols for a given backend.</p>
<p>flannel (CoreOS)</p> <p>flannel is a virtual network that gives a subnet to each host for use with container runtimes. Platforms like Google’s Kubernetes assume that each container (pod) has a unique, routable IP inside the cluster. The advantage of this model is that it reduces the complexity of doing port mapping.</p>

Product/Project (Company or Supporting Org.)

[Open vSwitch](#) (Linux Foundation)

A production quality, multilayer virtual switch licensed under the open source Apache 2.0 license. It is designed to enable massive network automation through programmatic extension, while still supporting standard management interfaces and protocols. In addition, it is designed to support distribution across multiple physical servers similar to VMware's vNetwork distributed vswitch or Cisco's Nexus 1000V.

[Longhorn](#) (Rancher Labs)

A distributed block storage system built using containers and microservices.

[Minio](#) (Minio)

Minio is an open source object storage server built for cloud applications and DevOps.

[Nuage Networks Virtualized Cloud Services \(VCS\)](#) (Nokia)

The datacenter and cloud networking framework of Nuage Networks Virtualized Services Platform (VSP).

[Nuage Networks Virtualized Services Platform \(VSP\)](#) (Nokia)

Provides software-defined networking capabilities for clouds of all sizes. It is implemented as a non-disruptive overlay for all existing virtualized and non-virtualized server and network resources. VSP is designed to work with Docker containers, Kubernetes and Mesos.

[OpenContrail](#) (Juniper Networks)

An Apache 2.0-licensed project that is built using standards-based protocols and provides all the necessary components for network virtualization: SDN controller, virtual router, analytics engine, and published northbound APIs. It has an extensive REST API to configure and gather operational and analytics data from the system.

[Portworx PX-Series](#) (Portworx)

A data layer for persistent storage that can be managed with Kubernetes.

[Project Calico](#) (Tigera)

Provides a scalable networking solution for connecting data center workloads (containers, VMs or bare metal). It uses a Layer 3 approach. Calico can be deployed without encapsulation or overlays to provide high performance at massive scales.

[Quay](#) (CoreOS)

A secure image registry that runs on your own servers.

[Redis](#) (Redis)

Redis is an in-memory database that persists on disk. The data model is key value, but many different kind of values are supported.

[Romana](#) (N/A)

A network and security automation solution for cloud-native applications. Romana automates the creation of isolated cloud-native networks and secures applications with a distributed firewall that applies access control policies consistently across all endpoints and services, wherever they run.

Product/Project (Company or Supporting Org.)

[Rook](#) (Quantum)

Storage for Kubernetes apps through persistent volumes.

[Tirame](#) (Aporeto)

An open-source library curated by Aporeto to provide segmentation for cloud-native applications.

[Twistlock](#) (Twistlock)

Twistlock is cloud-native cyber security for the modern enterprise. Advanced intelligence and machine learning capabilities automate policy creation and enforcement throughout the development life cycle. Native integration to leading CI/CD and orchestration tools provide security that enables innovation by not slowing development. Robust compliance checks and extensibility allow full control over your environment from developer workstations through to production.

[Vitess](#) (Google)

A database solution for scaling MySQL. It can run on Kubernetes.

[Weave Net](#) (Weaveworks)

Connects containers into a transparent, dynamic and resilient mesh. Weave Net creates a virtual network that connects Docker containers across multiple hosts and enables their automatic discovery.

DISCLOSURES

The following companies mentioned in this ebook are sponsors of The New Stack: Apcera, Aporeto, CA Technologies, Chef, Cloud Foundry Foundation, {code}, Containership, DigitalOcean, GoDaddy, HPE, InfluxData, Microsoft, OpenStack, Packet, PagerDuty, StackRox, The Linux Foundation, ThoughtWorks, Univa, VMware, Wercker.

Huawei is an advisory client of The New Stack.

A special thanks to [Joseph Jacks](#) for maintaining a spreadsheet of Kubernetes distributions.

