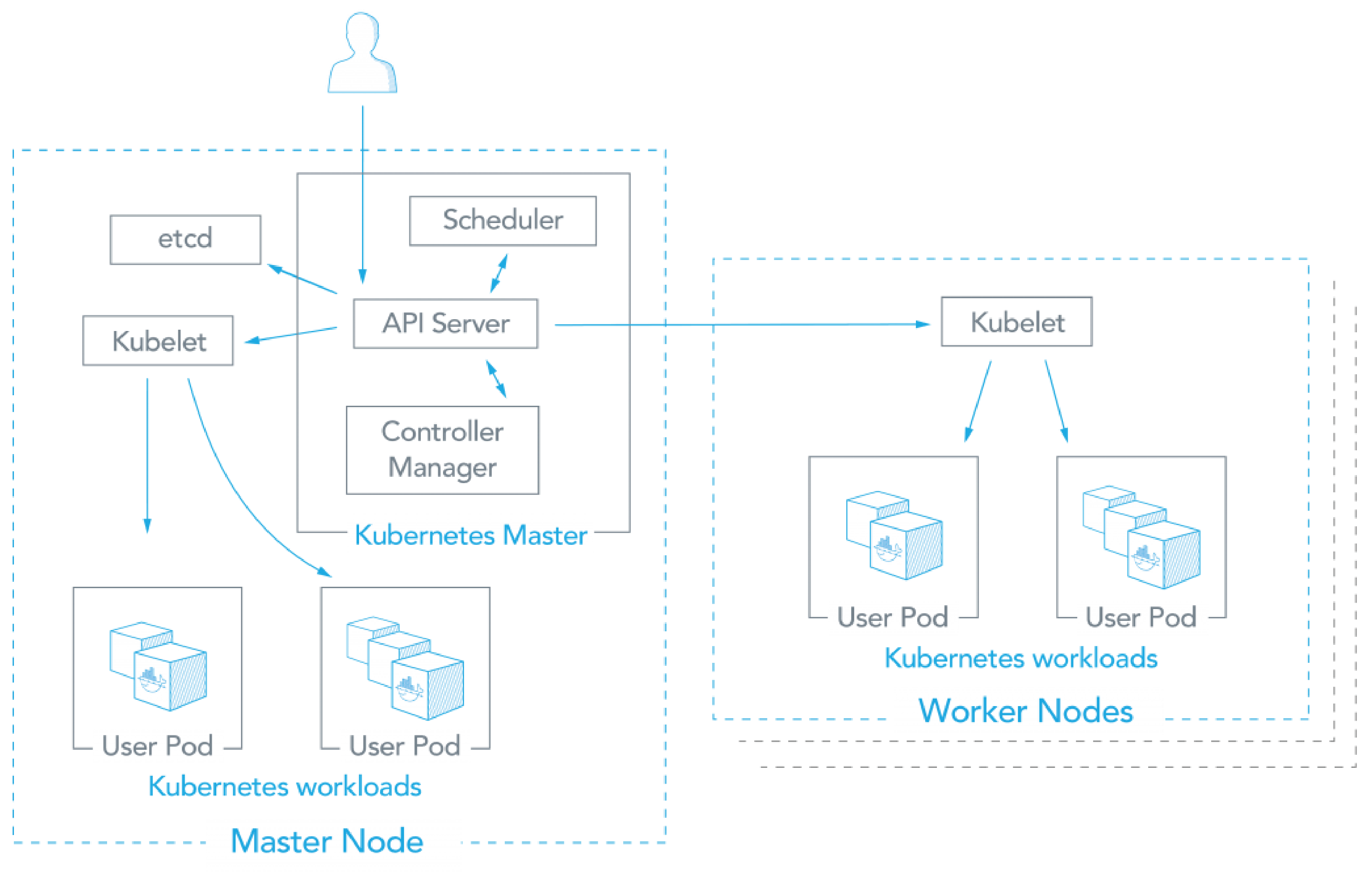
In this updated blog post we'll compare Kubernetes (versions 1.5.0 and later) with Docker Swarm, also referred to as Docker Engine running in Swarm mode (versions 1.12.0 and later). We'll walk you through high-level discussions of Kubernetes and Docker Swarm, and then compare them head-to-head.

Overview of Kubernetes

According to the Kubernetes website (https://kubernetes.io/), "Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications." Kubernetes was built by Google based on their experience running containers in production using an internal cluster management system called Borg (http://blog.kubernetes.io/2015/04/borg-predecessor-to-kubernetes.html) (sometimes referred to as Omega). The architecture for Kubernetes, which relies on this experience, is shown below:



API Server: management hub for Kubernetes
Scheduler: places a workload on the appropriate Node
Controller Manager: scales workloads up/down
etcd: stores configuration data which can be accessed by API Server

Kubelet: Receives pod specifications from API Server, updates Nodes
Master Node: places workloads on Nodes
Worker Nodes: receives requests from Master Nodes and dispatches them
User Pod: a group of containers with shared resources

As you can see from the figure above, there are a number of components associated with a Kubernetes cluster. The aster node places container workloads in user pods on worker nodes or itself. The other components include:

- etcd: This component stores configuration data which can be accessed by the Kubernetes Master's API Server using simple HTTP or JSON API.
- API Server: This component is the management hub for the Kubernetes master node. It facilitates communication between the various components, thereby maintaining cluster health.
- Controller Manager: This component ensures that the cluster's desired state matches the current state by scaling workloads up and down.
- Scheduler: This component places the workload on the appropriate node – in this case all workloads will be placed locally on your host.
- Kubelet: This component receives pod specifications from the API Server and manages pods running in the host.

The following list provides some other common terms associated with Kubernetes:

- Pods: Kubernetes deploys and schedules containers in groups called pods. Containers in a pod run on the same node and share resources such as filesystems, kernel namespaces, and an IP address.
- Deployments: These building blocks can be used to create and manage a group of pods. Deployments can be used with a service tier for scaling horizontally or ensuring availability.
- Services: Services are endpoints that can be addressed by name and can be connected to pods using label selectors. The service will automatically round-robin requests between pods. Kubernetes will set up a DNS

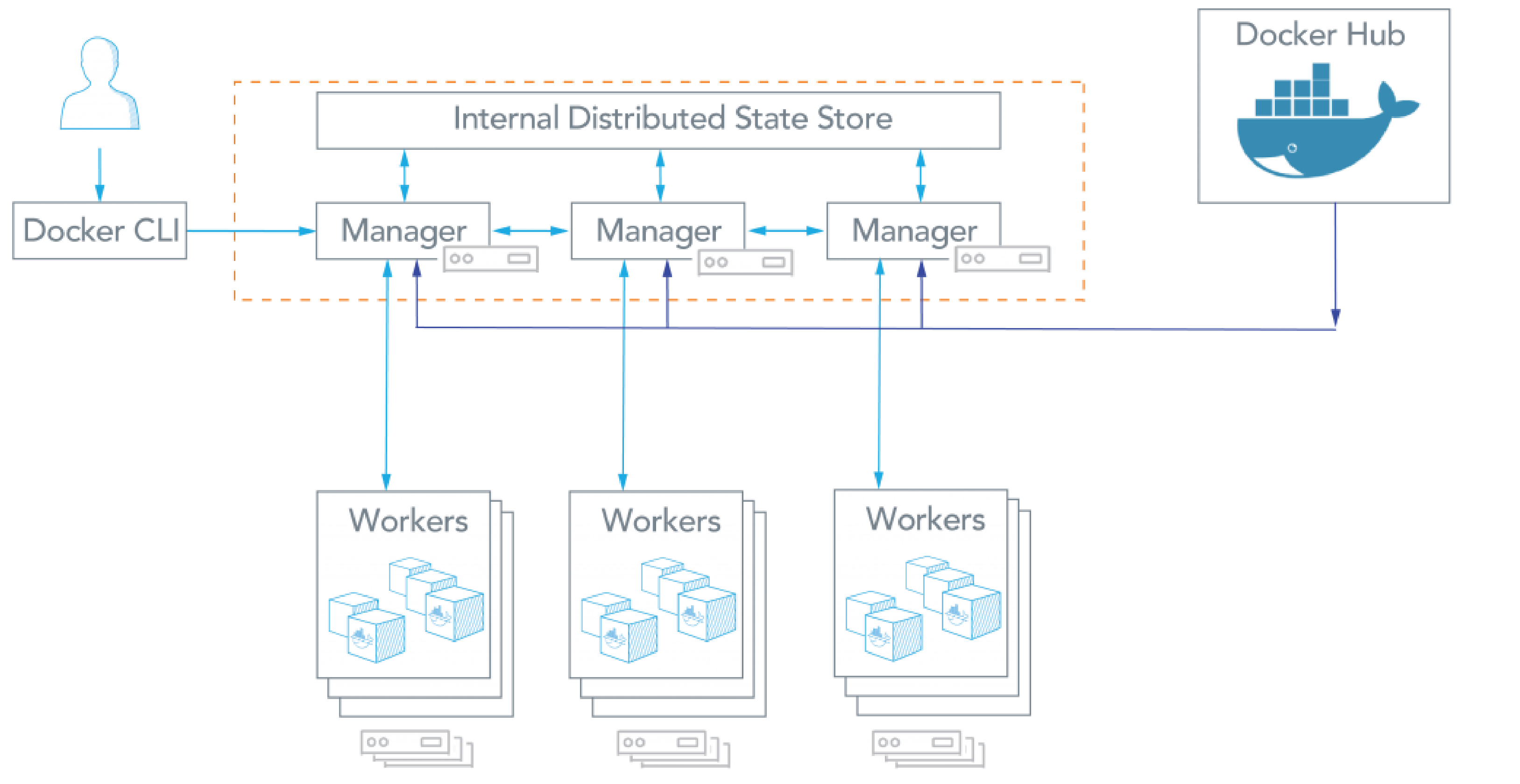
server for the cluster that watches for new services and allows them to be addressed by name. Services are the "external face" of your container workloads.

- Labels: Labels are key-value pairs attached to objects and can be used to search and update multiple objects as a single set.

Overview of Docker Swarm

Docker Engine v1.12.0 and later allow developers to deploy containers in Swarm mode. A Swarm cluster consists of Docker Engine deployed on multiple nodes. Manager nodes perform orchestration and cluster management. Worker nodes receive and execute tasks from the manager nodes.

A service, which can be specified declaratively, consists of tasks that can be run on Swarm nodes. Services can be replicated to run on multiple nodes. In the replicated services model, ingress load balancing and internal DNS can be used to provide highly available service endpoints. (Source: Docker Docs: Swarm mode (https://docs.docker.com/engine/swarm/key-concepts/))



Manager: a node that dispatches tasks
Worker: a node that executes tasks provided by a Manager
Internal Distributed Store: used to maintain cluster state

Docker CLI: User interacts with the swarm using Docker CLI, for example "docker service"
Docker Hub: contains repositories for downloading and sharing container images

As can be seen from the figure above, the Docker Swarm architecture consists of managers and workers. The user can declaratively specify the desired state of various services to run in the Swarm cluster using YAML files. Here are some common terms associated with Docker Swarm:

- Node: A node is an instance of a Swarm. Nodes can be distributed on-premises or in public clouds.
- Swarm: a cluster of nodes (or Docker Engines). In Swarm mode, you orchestrate services, instead of running container commands.
- Manager Nodes: These nodes receive service definitions from the user, and dispatch work to worker nodes. Manager nodes can also perform the duties of worker nodes.
- Worker Nodes: These nodes collect and run tasks from manager nodes.
- Service: A service specifies the container image and the number of replicas. Here is an example of a service command which will be scheduled on 2 available nodes:

```
# docker service create --replicas 2 --name mynginx nginx
```

- Task: A task is an atomic unit of a Service scheduled on a worker node. In the example above, two tasks would be scheduled by a master node on two worker nodes (assuming they are not scheduled on the Master itself). The two tasks will run independently of each other.

The next section compares the features of Kubernetes with Docker Swarm and the strengths/weaknesses of choosing one container orchestration solution over the other. You can also learn more about the container ecosystem by downloading the free Making Sense of the Container Ecosystem eBook (https://platform9.com/resources/compare-kubernetes-ebook/?utm_source=k8s_ds_compare).

(https://platform9.com/resources/compare-kubernetes-ebook/?utm_code=k8s_ds_compare)

Kubernetes vs. Docker Swarm: Technical Capabilities

Application Definition	
Applications can be deployed using a combination of pods, deployments, and services (or "microservices"). A pod is a group of co-located containers and is the atomic unit of a deployment. A deployment can have replicas across multiple nodes. A service is the "external face" of container workloads and integrates with DNS to round-robin incoming requests.	Applications can be deployed as services (or "microservices") in a Swarm cluster. Multi-container applications can specified using YAML files. Docker Compose can deploy the app. Tasks (an instance of a service running on a node) can be distributed across datacenters using labels. Multiple placement preferences can be used to distribute tasks further, for example, to a rack in a datacenter.
Application Scalability Constructs	
Each application tier is defined as a pod and can be scaled when managed by a deployment, which is specified in YAML. The scaling can be manual or automated. Pods can be used to run vertically integrated application stacks such as LAMP (Apache, MySQL, PHP) or ELK/Elastic (Elasticsearch, Logstash, Kibana), co-located and co-administered apps such as content management systems and apps for backups, checkpoint, compression, rotation, snapshotting.	Services can be scaled using Docker Compose YAML templates. Services can be global or replicated. Global services run on all nodes, replicated services run replicas (tasks) of the services across nodes. For example, A MySQL service with 3 replicas will run on a maximum of 3 nodes. Tasks can be scaled up or down, and deployed in parallel or in sequence.
High Availability	
Deployments allow pods to be distributed among nodes to provide HA, thereby tolerating application failures. Load-balanced services detect unhealthy pods and remove them. High availability of Kubernetes is supported. Multiple master nodes and worker nodes can be load balanced for requests from kubectl and clients. etcd can be clustered and API Servers can be replicated.	Services can be replicated among Swarm nodes. Swarm managers (https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/#manager-nodes) are responsible for the entire cluster and manage the resources of worker nodes. Managers use ingress load balancing to expose services externally. Swarm managers use Raft Consensus algorithm to ensure that they have consistent state information. An odd number of managers is recommended, and a majority of managers must be available for a functioning Swarm cluster (2 of 3, 3 of 5, etc.).

Load Balancing	
Pods are exposed through a service (http://kubernetes.io/v1.1/docs/user-guide/services.html), which can be used as a load-balancer within the cluster. Typically, an ingress (https://kubernetes.io/docs/concepts/services-networking/ingress/) is used for load balancing.	Swarm mode has a DNS component that can be used to distribute incoming requests to a service name. Services can run on ports specified by the user or can be assigned automatically.
Auto-scaling for the Application	
Auto-scaling using a simple number-of-pods target is defined declaratively using deployments (https://kubernetes.io/docs/concepts/workloads/controllers/deployment/). CPU-utilization-per-pod target is available. Other targets are on the roadmap.	Not directly available. For each service, you can declare the number of tasks you want to run. When you manually scale up or down, the Swarm manager automatically adapts by adding or removing tasks.
Rolling Application Upgrades and Rollback	
The deployment controller supports both "rolling-update" and "recreate" strategies (https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#strategy). Rolling updates can specify maximum number of pods unavailable or maximum number running during the process.	At rollout time, you can apply rolling updates to services (https://docs.docker.com/engine/swarm/swarm-tutorial/rolling-update/#apply-rolling-updates-to-a-service). The Swarm manager lets you control the delay between service deployment to different sets of nodes, thereby updating only 1 task at a time.
Health Checks	
Health checks of two kinds (http://kubernetes.io/v1.1/docs/user-guide/production-pods.html#liveness-and-readiness-probes-aka-health-checks): liveness (is app responsive) and readiness (is app responsive, but busy preparing and not yet able to serve) Out-of-the-box K8S provides a basic logging mechanism (https://kubernetes.io/docs/concepts/cluster-administration/logging/) to pull aggregate logs for a set of containers that make up a pod.	Docker Swarm health checks (https://forums.docker.com/t/how-does-the-health-check-work-in-a-swarm/22151) are limited to services. If a container backing the service does not come up (running state), a new container is kicked off. Users can embed health check (https://ryaneschinger.com/blog/using-docker-native-health-checks/) functionality into their Docker images using the HEALTHCHECK instruction.
Storage	

Two storage APIs: The first provides abstractions for individual storage backends (http://kubernetes.io/docs/user-guide/volumes/) (e.g. NFS, AWS EBS, ceph, flocker). The second provides an abstraction for a storage resource request (http://kubernetes.io/docs/user-guide/persistent-volumes/) (e.g. 8 Gb), which can be fulfilled with different storage backends. Modifying the storage resource used by the Docker daemon on a cluster node requires temporarily removing the node from the cluster. Kubernetes offers several types of persistent volumes with block or file support. Examples include iSCSI, NFS, FC, Amazon Web Services, Google Cloud Platform, and Microsoft Azure. The emptyDir volume is non-persistent and can used to read and write files with a container.	Docker Engine and Swarm support mounting volumes (https://docs.docker.com/engine/tutorials/dockervolumes/) into a container. Shared filesystems, including NFS, iSCSI, and fibre channel, can be configured nodes. Plugin options include Azure, Google Cloud Platform, NetApp, Dell EMC, and others.
Networking	
The networking model is a flat network, enabling all pods to communicate with one another. Network policies specify how pods communicate with each other. The flat network is typically implemented as an overlay. The model requires two CIDRs: one from which pods get an IP address, the other for services.	Node joining a Docker Swarm cluster creates an overlay network for services that span all of the hosts in the Swarm and a host only Docker bridge network for containers. By default, nodes in the Swarm cluster encrypt overlay control and management traffic between themselves. Users can choose to encrypt container data traffic when creating an overlay network by themselves.
Service Discovery	

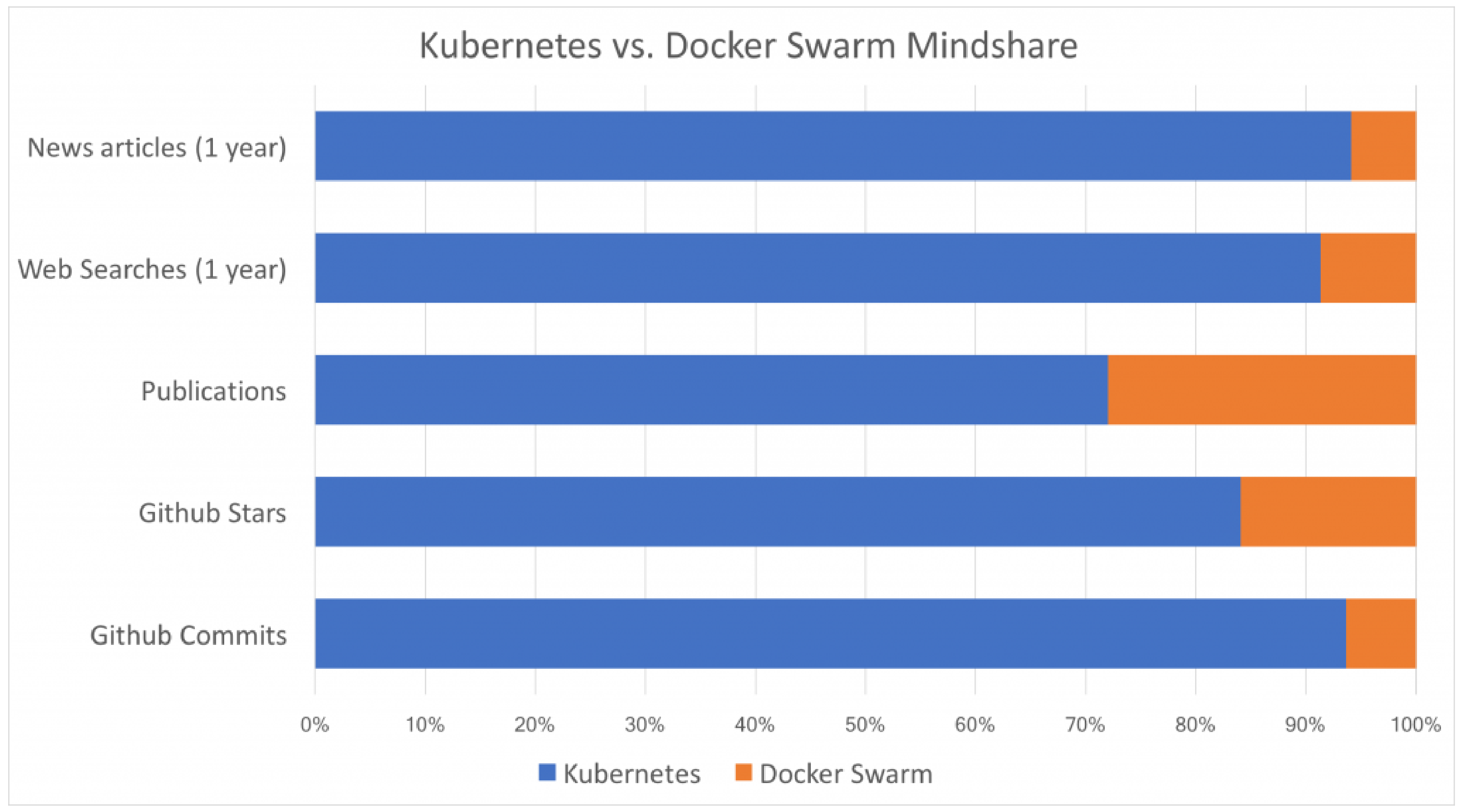
Services can be found using environment variables or DNS. Kubelet adds a set of environment variables when a pod is run. Kubelet supports simple {SVCNAME_SERVICE_HOST} AND {SVCNAME_SERVICE_PORT} variables, as well as Docker links compatible variables. DNS Server is available as an addon. For each Kubernetes Service, the DNS Server creates a set of DNS records. If DNS is enabled in the entire cluster, pods will be able to use Service names that automatically resolve.	Swarm Manager node assigns each service a unique DNS name and load balances running containers (https://docs.docker.com/engine/swarm/networking/#use-dns-round-robin-for-a-service). Requests to services are load balanced to the individual containers via the DNS server embedded in the Swarm. Docker Swarm comes with multiple discovery backends (https://docs.docker.com/swarm/discovery/#docker-swarm-discovery): • Docker Hub as a hosted discovery service is intended to be used for dev/test. Not recommended for production. • A static file or list of nodes can be used as a discovery backend. The file must be stored on a host that is accessible from the Swarm Manager. You can also provide a node list as an option when you start Swarm.
Performance and scalability	
With the release of 1.6, Kubernetes scales to 5,000-node clusters (http://blog.kubernetes.io/2017/03/scalability-updates-in-kubernetes-1.6.html). Kubernetes scalability is benchmarked against the following Service Level Objectives (SLOs): • API responsiveness: 99% of all API calls return in less than 1s. • Pod startup time: 99% of pods and their containers (with pre-pulled images) start within 5s.	According to the Docker's blog post on scaling Swarm clusters (https://blog.docker.com/2015/11/scale-testing-docker-swarm-30000-containers/), Docker Swarm has been scaled and performance tested up to 30,000 containers and 1,000 nodes with 1 Swarm manager.

Synopsis

Pros of Kubernetes	
• Based on extensive experience running Linux containers at Google. Deployed at scale more often among organizations. Kubernetes is also backed by enterprise offerings from both Google (GKE) and RedHat (OpenShift). • Can overcome constraints of Docker and Docker API. • Autoscaling based on factors such as CPU utilization. • Largest community among container orchestration tools. Over 50,000 commits and 1200 contributors.	• Does not have as much experience with production deployments at scale. • Limited to the Docker API's capabilities. • Services can be scaled manually. • Smaller community and project. Over 3,000 commits and 160 contributors.
Cons of Kubernetes	
• Do-it-yourself installation can be complex, but flexible. Deployment tools include kubeadm, kops, kargo, and others. Further details in Kubernetes Deployment Guide (https://platform9.com/resources/kubernetes-deployment-models/?utm_source=k8s_ds_compare). • Uses a separate set of tools for management, including kubectl CLI.	• Deployment is simpler and Swarm mode is included in Docker Engine. • Integrates with Docker Compose and Docker CLI – native Docker tools. Many of the Docker CLI commands will work with Swarm. Easier learning curve.
Common Features	
• Open source projects. Anyone can contribute using the Go programming language. • Various storage options. • Networking features such as load balancing and DNS. • Logging and Monitoring add-ons. These external tools include Elasticsearch/Kibana (ELK), sysdig, cAdvisor, Heapster/Grafana/InfluxDB. ◦ Logging and Monitoring for Kubernetes (https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/) ◦ Logging and Monitoring for Docker Swarm (https://www.google.com/search?q=docker+swarm+logging+monitoring&rlz=1C5CHFA_enUS740US741&oq=docker+swarm+logging+monitoring&aqs=chrome..69i57j69i60.12431j0j8&sourceid=chrome&ie=UTF-8)	

Takeaways

Kubernetes has been deployed more widely than Docker Swarm, and is validated by Google. The popularity of Kubernetes is evident in the chart, which shows Kubernetes compared with Swarm on five metrics: news articles and scholarly publications over the last year, Github stars and commits, and web searches on Google.



Kubernetes leads on all metrics: when compared with Docker Swarm, Kubernetes has over 80% of the mindshare for news articles, Github popularity, and web searches. However, there is general consensus that Kubernetes is also more complex to deploy and manage. The Kubernetes community has tried to mitigate this drawback by offering a variety of deployment options, including Minikube and kubeadm. Platform9's Managed Kubernetes product also fills this gap by letting organizations focus on deploying microservices on Kubernetes, instead of managing and upgrading a highly available Kubernetes deployment. Further details on these and other deployment models for Kubernetes can be found in The Ultimate Guide to Deploy Kubernetes (https://platform9.com/resources/kubernetes-deployment-models/?utm_source=k8s_ds_compare).

In a follow-up blogs, we'll compare Kubernetes with Amazon ECS and Mesos. Meanwhile, please feel free to take a guided tour of Platform9's Managed Kubernetes using a Sandbox (https://platform9.com/sandbox/?utm_source=k8s_ds_compare), and share any comments below.

(https://platform9.com/resources/compare-kubernetes-ebook/?utm_code=k8s_ds_compare)