eBook

Making Sense of the Container Ecosystem

Comparing Kubernetes with Docker Swarm, Apache Mesos and AWS EC2 Container Service

PLATFORM9

Introduction to Container Orchestration

Containers make it very easy to run cloud-native applications on physical or virtual infrastructure. They are lighter weight compared to VMs and make more efficient use of the underlying infrastructure. Containers are meant to make it easy to turn apps on and off to meet fluctuating demand and move apps seamlessly between different environments or even clouds. While the container runtime APIs meet the needs of managing one container on one host, they are not suited to manage multiple containers deployed on multiple hosts. This is where we need to look at container orchestration tools.

Container Orchestration tools can manage complex, multi-container apps deployed on a cluster of machines. These tools can treat an entire cluster as a single entity for deployment and management. Container orchestration tools can automate all aspects from initial placement, scheduling and deployment to updates and health monitoring functions that support scaling and failover. Container orchestration tools provide built-in support for a number of painpoints developers face while building production applications, such as service discovery, load balancing, rolling upgrades, health monitoring, auto-scaling etc.

Capabilities of Container Orchestration Tools

Here are some of the main capabilities that a modern container orchestration platform will typically provide:

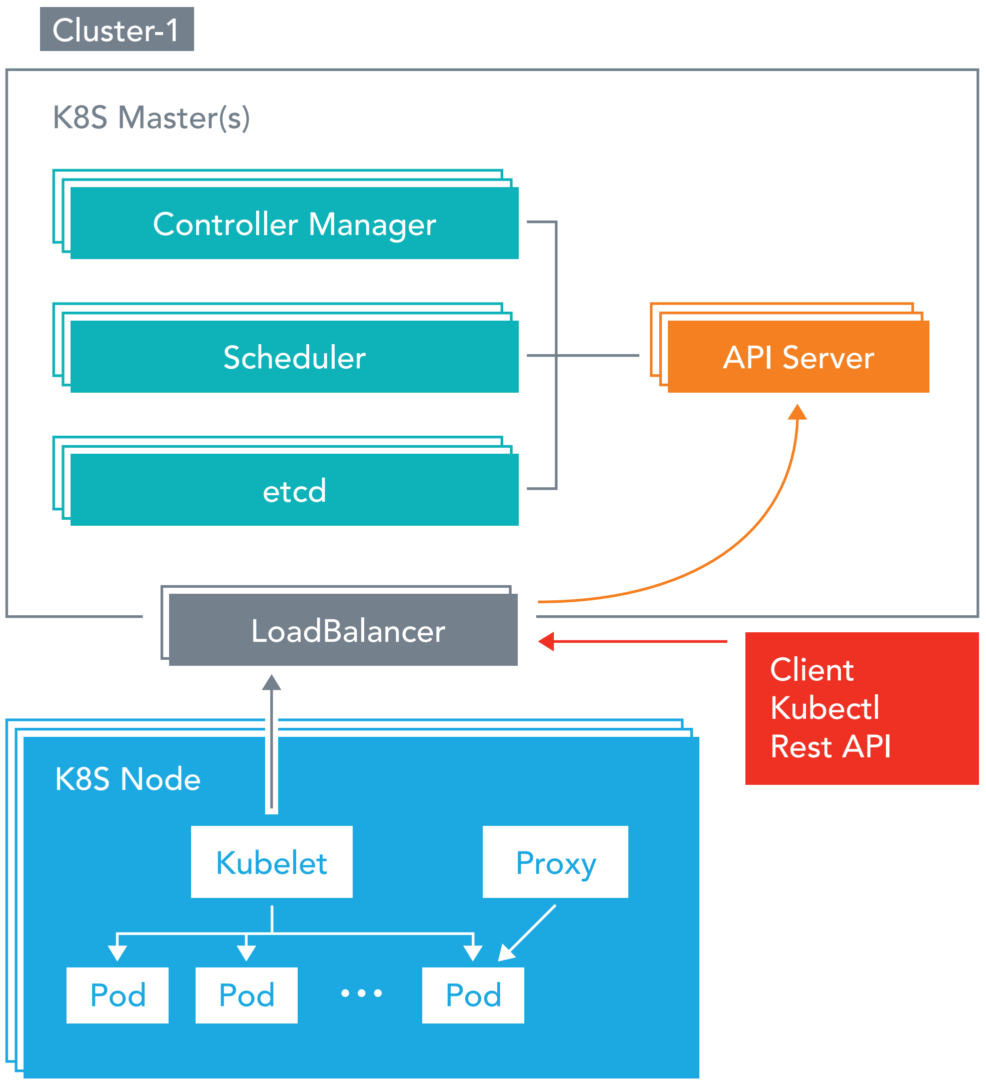
- **Provisioning**
  Container orchestration tools can provision or schedule containers within the cluster and launch them. As part of this, the tool will determine the right placement for the containers by selecting an appropriate host based on the specified constraints such as resource requirements, location affinity etc. The underlying goal is to increase utilization of the available resources. Most tools will be agnostic to the underlying infrastructure provider and, in theory, should be able to move containers across environments and clouds.
- **Configuration-as-text**
  Container orchestration tools can load the application blueprint from a schema defined in YAML or JSON. These are popular languages to define infrastructure-as-code similar to OpenStack Heat templates, Puppet Manifests or Chef recipes. Defining the blueprint in this manner makes it easy for DevOps teams to edit, share and version the configurations and provide repeatable deployments across development, testing and production.
- **Monitoring**
  Container orchestration tools will track and monitor the health of the containers and hosts in the cluster. If a container crashes, a new one can be spun up quickly. If a host fails, the tool will restart the failed containers on another host. It will also run specified health checks at the appropriate frequency and update the list of available nodes based on the results. In short, the tool will ensure that the current state of the cluster matches the configuration specified.
- **Rolling Upgrades and Rollback**
  One of the most desired ability is for the orchestration tool to perform 'rolling upgrades' of the application where a new version is applied incrementally across the cluster. Traffic is routed appropriately as containers go down temporarily to receive the update. A rolling update guarantees a minimum number of "ready" containers at any point, so that all old containers are not replaced if there aren't enough healthy new containers to replace them. If, however, the new version doesn't perform as expected then the orchestration tool can also rollback the applied change.
- **Policies for Placement, Scalability etc.**
  Container orchestration tools provide a way to define policies for host placement, security, performance and high availability. When configured correctly, container orchestration platforms can enable organizations to deploy and operate containerized application workloads in a secure, reliable and scalable way. For example, an application can be scaled up automatically based on CPU usage of the containers.

- **Service Discovery**
  Since containers encourage a microservices based architecture, service discovery becomes a critical function and is provided in different ways by container orchestration platforms e.g. DNS or proxy-based etc. For example, a web application front-end dynamically discovering another microservice or a database.
- **Ease of Administration**
  Container orchestration tools themselves should be easy to deploy, configure and setup for Administrators. They should connect to existing IT tools for SSO, RBAC etc. An extensible architecture will connect to external systems such as local or cloud storage, networking systems etc.

This was a brief overview of the importance of choosing the right container orchestration tool to deploy and manage cloud-native applications. Below we'll introduce Kubernetes, Mesos, Docker Swarm and Amazon EC2 Container Service. We'll also compare Kubernetes with each of the other tools.

Kubernetes

According to the Kubernetes website – "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 over the last decade. See below for a Kubernetes architecture diagram and the following explanation.



The major components in a Kubernetes cluster are:

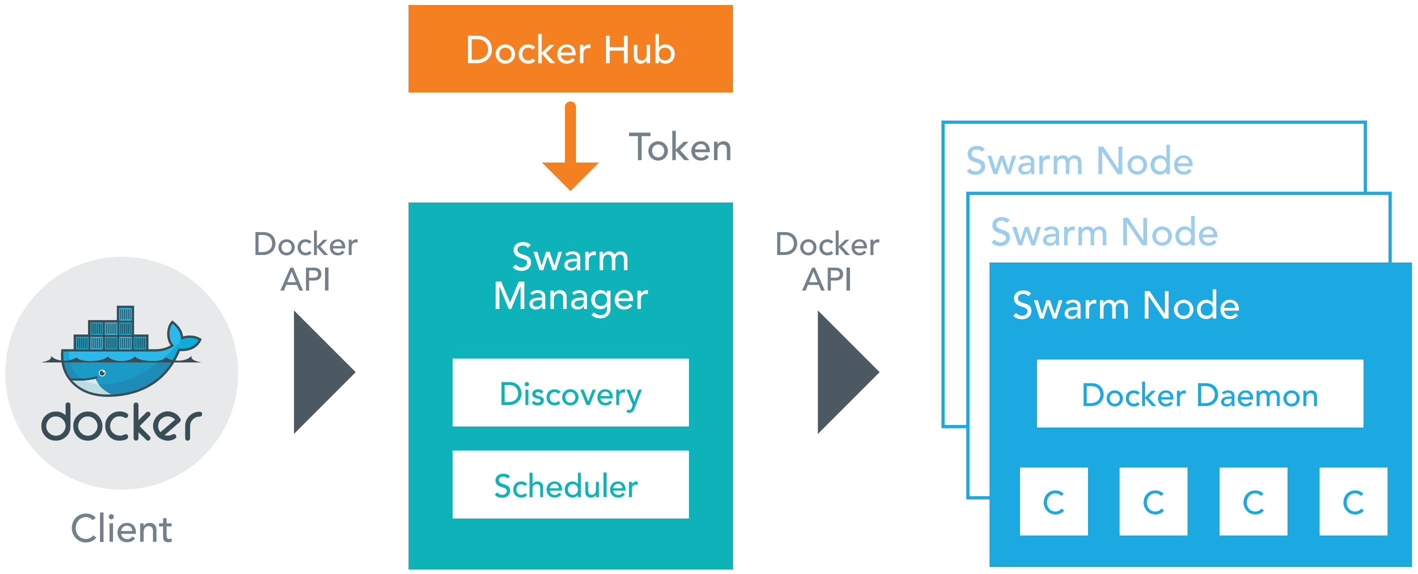
- **Pods** – Kubernetes deploys and schedules containers in groups called pods. A pod will typically include 1 to 5 containers that collaborate to provide a service.
- **Flat Networking Space** – The default network model in Kubernetes is flat and permits all pods to talk to each other. Containers in the same pod share an IP and can communicate using ports on the localhost address.
- **Labels** – Labels are key-value pairs attached to objects and can be used to search and update multiple objects as a single set.
- **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 the pods. Kubernetes will set up a DNS server for the cluster that watches for new services and allows them to be addressed by name.
- **Replication Controllers** – Replication controllers are the way to instantiate pods in Kubernetes. They control and monitor the number of running pods for a service, improving fault tolerance.

Docker Swarm

Docker v1.12 includes a swarm mode in Docker Engine for natively managing a cluster of Docker Engines called a warm. The Docker CLI can be used to create a swarm, deploy application services to a swarm, and manage swarm behavior. This is backwards compatible with the previous Docker Swarm that was available as a stand-alone option prior to v1.12. While both options are still available, Docker recommends using the Swarm mode going forward.

The comparison in this section will mostly apply to either of the options available for Docker Swarm. Swarm uses the standard Docker API, so normal Docker run commands can be used to launch containers and Swarm will take care of selecting an appropriate host to run the container on.

Other tools that use the Docker API, e.g. Docker Compose, worked with the stand-alone Swarm without any changes, but are still not integrated with the Swarm mode.



Each host in a Swarm cluster runs a Swarm agent and one host runs a Swarm manager. The manager will orchestrate and schedule containers on the hosts. Similar to other container orchestration tools, a Discovery service will find and add new hosts to the cluster. Third-party tools like Consul, ZooKeeper, etcd are required to ensure high availability and failover to a secondary Swarm manager. The table below gives a detailed comparison of Swarm features and compares them with Kubernetes.

Compare Kubernetes v/s Docker Swarm

The figure below shows Kubernetes clusters in a multi-master configuration. Each cluster runs the Master node services in a highly available manner. Similar to the OpenStack example above, both clusters have to be configured for security, backup and Single Sign On services. In a production environment, the clusters will have to be continuously monitored for health and performance, and updated regularly with patches and new versions of Kubernetes.

Type's of Workloads	**Cloud Native applications**	**Cloud Native applications**
Application Definition	A combination of Pods, Replication Controllers, Replica Sets, Services and Deployments. As explained in the overview above, a pod is a group of co-located containers; the atomic unit of deployment. Pods do not express dependencies among individual containers within them. Containers in a single Pod are guaranteed to run on a single Kubernetes node.	Apps defined in Docker Compose can be deployed on a Swarm cluster.
Application Scalability constructs	Each application tier is defined as a pod and can be scaled when managed by a Deployment or Replication Controller. The scaling can be manual or automated.	Docker CLI can be used to scale the number of services in the swarm. $docker service scale <SERVICE-ID>=<NUMBER-OF-TASKS> https://docs.docker.com/engine/swarm/swarm-tutorial/scale-service/
High Availability	Pods are distributed among Worker Nodes. Services also HA by detecting unhealthy pods and removing them.	Containers are distributed among Swarm Nodes. The Swarm manager is responsible for the entire cluster and manages the resources of multiple Docker hosts at scale. To ensure the Swarm manager is highly available, a single primary manager instance and multiple replica instances must be created. Requests issued on a replica are automatically proxied to the primary manager. If a primary manager fails, tools like Consul, ZooKeeper or etcd will pick a replica as the new manager.

Continued on following page.

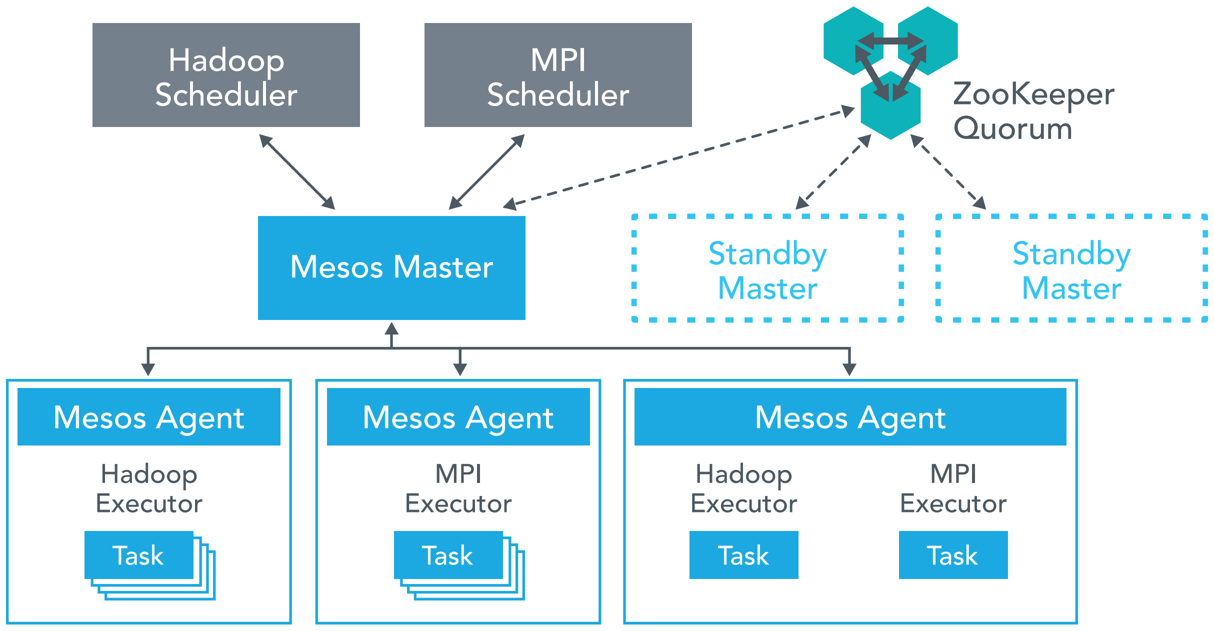
Type's of Workloads	Cloud Native applications	Cloud Native applications
Load Balancing	Pods are exposed through a Service, which can be a load balancing.	The swarm manager uses ingress load balancing to expose the services you want to make available externally to the swarm. The swarm manager can automatically assign the service a Published Port or users can configure a Published Port for the service. External components, such as cloud load balancers, can access the service on the Published Port of any node in the cluster whether or not the node is currently running the task for the service. All nodes in the swarm route ingress connections to a running task instance. Swarm mode has an internal DNS component that automatically assigns each service in the swarm a DNS entry. The swarm manager uses internal load balancing to distribute requests among services within the cluster based upon the DNS name of the service.
Auto-scaling for the Application	Auto-scaling using a simple number-of-pods target is defined declaratively with the API exposed by Replication Controllers. CPU-utilization-per-pod target is available as of v1.1 in the Scale subresource. Other targets are on the roadmap.	For each service, you can declare the number of tasks you want to run. When you scale up or down, the swarm manager automatically adapts by adding or removing tasks to maintain the desired state.
Rolling Application Upgrades and Rollback	"Deployment" model supports strategies, but one similar to Mesos is planned for the future. Health checks test for liveness i.e. is app responsive.	At rollout time, you can apply service updates to nodes incrementally. The swarm manager lets you control the delay between service deployment to different sets of nodes. If anything goes wrong, you can roll-back a task to a previous version of the service.
Logging and monitoring	Health checks of two kinds: liveness (is app responsive) and readiness (is app responsive, but busy preparing and not yet able to serve) Logging: Container logs shipped to Elasticsearch/Kibana (ELK) service deployed in cluster Resource usage monitoring: Heapster/Grafana/Influx service deployed in cluster Logging/monitoring add-ons are part of official project Sysdig Cloud integration	Logging: Can ship logs to ELK deployed in cluster Monitoring: Can use external tools, e.g. Riemann

Continued on following page.

Type's of Workloads	Cloud Native applications	Cloud Native applications
Storage	Two storage APIs: The first provides abstractions for individual storage backends (e.g. NFS, AWS EBS, ceph,- flocker). The second provides an abstraction for a storage resource request (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.	Docker Engine and Swarm support mounting volumes into a container. A volume is stored locally by default. Volume plugins (e.g.flocker) mount volumes on networked storage (e.g., AWS EBS, Cinder, Ceph).
Networking	The networking model lets any pod can communicate with other pods and with any service. The model requires two networks (one for pods, the other for services) Neither network is assumed to be (or needs to be) reachable from outside the cluster. The most common way of meeting this requirement is to deploy an overlay network on the cluster nodes.	You can specify an overlay network for your services. The swarm manager automatically assigns addresses to the containers on the overlay network when it initializes or updates the application.
Service Discovery	Pods discover services using intra-cluster DNS	Swarm manager node assigns each service a unique DNS name and load balances running containers. You can query every container running in the swarm through a DNS server embedded in the swarm.
Performance and scalability	With the release of 1.2, Kubernetes now supports 1000-node clusters. 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 Swarm website, Swarm is production ready and tested to scale up to one thousand (1,000) nodes and fifty thousand (50,000) containers with no performance degradation in spinning up incremental containers onto the node cluster. Check out these other blogs on the topic- https://blog.docker.com/2015/11/scale-testing-docker-swarm-30000-containers/ https://medium.com/on-docker/evaluating-container-platforms-at-scale-5e7b44d-93f2c#.sblte6chl

Apache Mesos (+Marathon)

Apache Mesos is an open-source cluster manager designed to scale to very large clusters, from hundreds to thousands of hosts. Mesos supports diverse kinds of workloads such as Hadoop tasks, cloud native applications etc. The architecture of Mesos is designed around high-availability and resilience.



Credit: http://mesos.apache.org/documentation/latest/architecture/

The major components in a Mesos cluster are:

- Mesos Agent Nodes – Responsible for actually running tasks. All agents submit a list of their available resources to the master.
- Mesos Master – The master is responsible for sending tasks to the agents. It maintains a list of available resources and makes "offers" of them to frameworks e.g. Hadoop. The master decides how many resources to offer based on an allocation strategy. There will typically be stand-by master instances to take over in case of a failure.
- ZooKeeper – Used in elections and for looking up address of current master. Multiple instances of ZooKeeper are run to ensure availability and handle failures.
- Frameworks – Frameworks co-ordinate with the master to schedule tasks onto agent nodes. Frameworks are composed of two parts-
  - the executor process runs on the agents and takes care of running the tasks and
  - the scheduler registers with the master and selects which resources to use based on offers from the master.

There may be multiple frameworks running on a Mesos cluster for different kinds of task. Users wishing to submit jobs interact with frameworks rather than directly with Mesos.

In the figure above, a Mesos cluster is running alongside the Marathon, framework as the scheduler. The Marathon scheduler uses ZooKeeper to locate the current Mesos master which it will submit tasks to. Both the Marathon scheduler and the Mesos master have standbys ready to start work should the current master become unavailable.

Marathon, created by Mesosphere, is designed to start, monitor and scale long-running applications, including cloud native apps. Clients interact with Marathon through a REST API. Other features include support for health checks and an event stream that can be used to integrate with load-balancers or for analyzing metrics.

Compare Kubernetes v/s Mesos(+Marathon)

	Kubernetes	**Mesos**
Types of Workloads	Cloud Native applications	Support for diverse kinds of workloads such as big data, cloud native apps etc.
Application Definition	A combination of Pods, Replication Controllers, Replica Sets, Services and Deployments. As explained in the overview above, a pod is a group of co-located containers; the atomic unit of deployment. Pods do not express dependencies among individual containers within them. Containers in a single Pod are guaranteed to run on a single Kubernetes node.	"Application Group" models dependencies as a tree of groups. Components are started in dependency order. Colocation of group's containers on same Mesos slave is not supported. A Pod abstraction is on roadmap, but not yet available.
Application Scalability constructs	Each application tier is defined as a pod and can be scaled when managed by a Deployment or Replication Controller. The scaling can be manual or automated.	Possible to scale an individual group, its dependents in the tree are also scaled.
High Availability	Pods are distributed among Worker Nodes. Services also HA by detecting unhealthy pods and removing them.	Applications are distributed among Slave Nodes.
Load Balancing	Pods are exposed through a Service, which can be a load balancer.	Application can be reached via Mesos-DNS, which can act as a rudimentary load balancer. There are other options like 1- Minuteman https://github.com/dcos/minuteman and Marathon load balancer https://mesosphere.com/blog/2015/12/04/dcos-marathon-lb/
Auto-scaling for the Application	Auto-scaling using a simple number-of-pods target is defined declaratively with the API exposed by Replication Controllers. CPU-utilization-per-pod target is available as of v1.1 in the Scale subresource. Other targets are on the roadmap.	Load-sensitive autoscaling available as a proof-of-concept application. Rate-sensitive autoscaling available for Mesosphere's enterprise customers. Rich metric-based scaling policy.
Rolling Application Upgrades and Rollback	"Deployment" model supports strategies, but one similar to Mesos is planned for the future. Health checks test for liveness i.e. is app responsive.	"Rolling restarts" model uses application-defined minimum Health Capacity (ratio of nodes serving new/old application) "Health check" hooks consume a "health" API provided by the application itself.

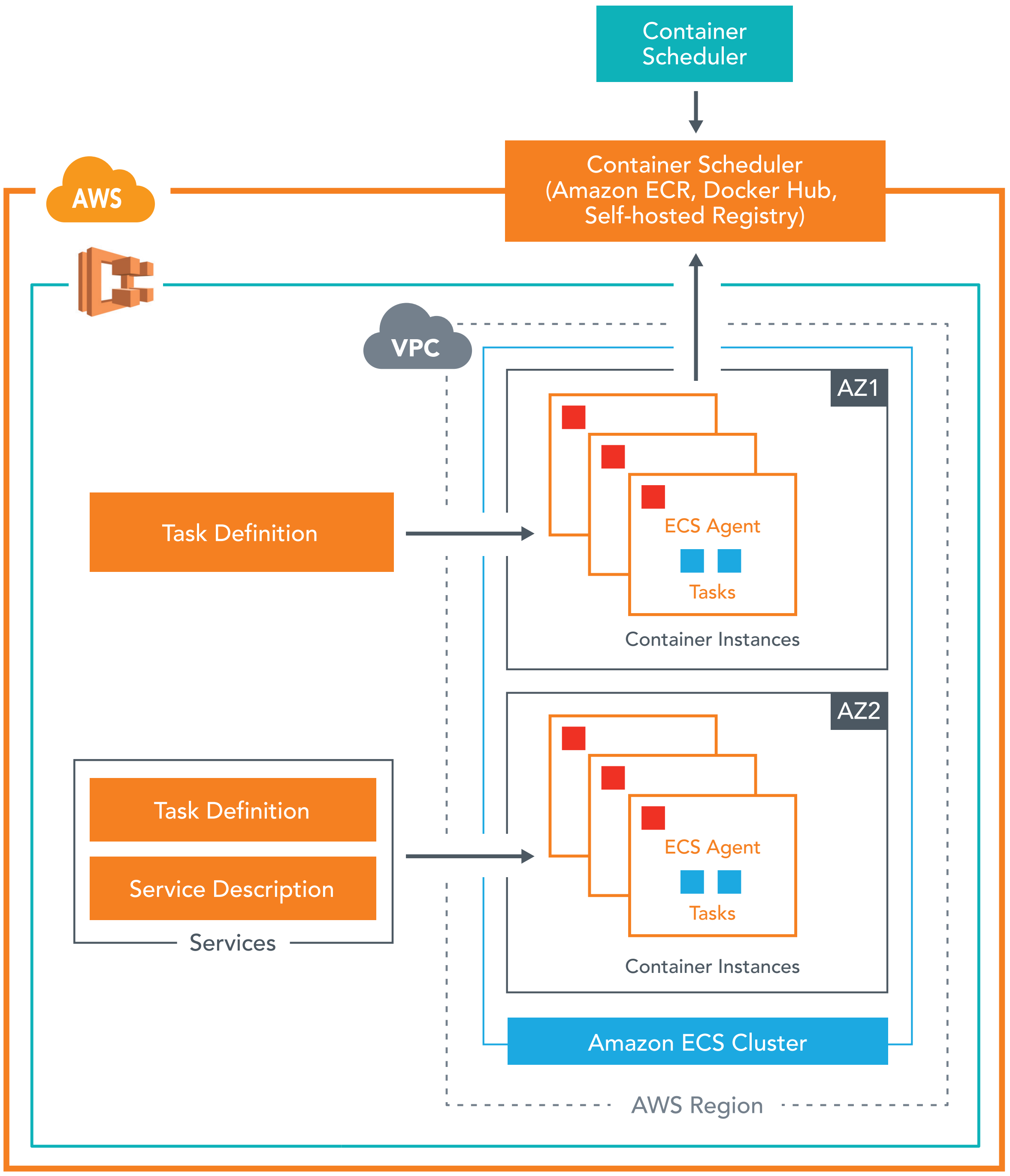
Continued on following page.

	Kubernetes	Mesos
Logging and monitoring	Health checks of two kinds: liveness (is app responsive) and readiness (is app responsive, but busy preparing and not yet able to serve). Logging: Container logs shipped to Elasticsearch/Kibana (ELK) service deployed in cluster. Resource usage monitoring: Heapster/Grafana/Influx service deployed in cluster. Logging/monitoring add-ons are part of official project . Sysdig Cloud integration	Logging: Can use ELK Monitoring: Can use external tools
Storage	Two storage APIs: The first provides abstractions for individual storage backends (e.g. NFS, AWS EBS, Ceph, Flocker). The second provides an abstraction for a storage resource request (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	A Marathon container can use persistent volumes, but such volumes are local to the node where they are created, so the container must always run on that node. An experimental flocker integration supports persistent volumes that are not local to one node.
Networking	The networking model lets any pod can communicate with other pods and with any service. The model requires two networks (one for pods, the other for services) . Neither network is assumed to be (or needs to be) reachable from outside the cluster. The most common way of meeting this requirement is to deploy an overlay network on the cluster nodes.	Marathon's Docker integration facilitates mapping container ports to host ports, which are a limited resource. Mesos also has Ip-per-container support https://mesosphere.com/blog/2015/12/02/ip-per-container-mesos/
Service Discovery	Pods discover services using intra-cluster DNS.	Containers can discover services using DNS or reverse proxy.
Performance and scalability	With the release of 1.2, Kubernetes now supports 1000-node clusters. 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.	Among the other tools, Mesos has been used more often for larger clusters. Twitter has a tweaked version of Mesos spanning more than 80,000 nodes. Mesos can run LXC or Docker containers directly from the Marathon framework or it can fire up Kubernetes or Docker Swarm (the Docker-branded container manager) and let them do it.

AWS EC2 Container Service (ECS)

According to the AWS ECS webpage, Amazon EC2 Container Service (ECS) is a highly scalable, high performance container management service that supports Docker containers and allows you to easily run applications on a managed cluster of Amazon EC2 instances. Amazon ECS eliminates the need for you to install, operate, and scale your own cluster management infrastructure.

Note that the containers managed by ECS will be exclusively run on AWS EC2 instances. There's no support for containers to run on infrastructure outside of EC2, whether physical infrastructure or other clouds. In addition, EC2 instances must be created prior to requesting ECS to bring up containers on those instances. This is a big difference compared to other container orchestration solutions, which do not lock the user into a particular infrastructure provider. The advantage, of course, is the ability to work with all the other AWS services like Elastic Load Balancers, CloudTrail, CloudWatch etc.



Credit: http://docs.aws.amazon.com/AmazonECS/latest/developerguide

The major components in Amazon ECS are:

- **Task Definition**: The task definition is a text file, in JSON format, describing the containers that together form an application. Task definitions specify various parameters for the application e.g. container image repositories, ports, storage etc.
- **Tasks and Scheduler**: A task is an instance of a task definition, created at runtime on a container instance within the cluster. The task scheduler is responsible for placing tasks on container instances.
- **Service**: A service is a group of tasks that are created and maintained as instances of a task definition. The scheduler maintains the desired count of tasks in the service. A service can optionally run behind a load balancer. The load balancer distributes traffic across the tasks that are associated with the service.
- **Cluster**: A cluster is a logical grouping of EC2 instances on which ECS tasks are run.
- **Container Agent**: The container agent runs on each EC2 instance within an ECS cluster. The agent sends telemetry data about the instance's tasks and resource utilization to Amazon ECS. It will also start and stop tasks based on requests from ECS.
- **Image Repository**: Amazon ECS downloads container images from container registries, which may exist within or outside of AWS, such as a accessible private Docker registry or Docker Hub.

Compare Kubernetes v/s ECS

	Kubernetes	**Amazon ECS**
Deployment Infrastructure	Physical H/W, Virtual Infra or public clouds.	Only on AWS EC2 instances.
Application Definition	A combination of Pods, Replication Controllers, Replica Sets, Services and Deployments. As explained in the overview above, a pod is a group of co-located containers; the atomic unit of deployment. Pods do not express dependencies among individual containers within them. Containers in a single Pod are guaranteed to run on a single Kubernetes node.	Application can span multiple task definitions by combining related containers into their own task definitions, each representing a single component. Task definitions group the containers that are used for a common purpose, and separate the different components into multiple task definitions.
Application Scalability constructs	Each application tier is defined as a pod and can be scaled when managed by a Deployment or Replication Controller. The scaling can be manual or automated.	Task instances can be scaled by updating their task definitions or the underlying EC2 instances can be scaled based on monitoring alerts.
High Availability	Pods are distributed among Worker Nodes. Services also HA by detecting unhealthy pods and removing them.	By using AWS Availability zones and defining placement policies in the Service requirements.
Load Balancing	Pods are exposed through a Service, which can be a load balancer.	Amazon ECS can optionally be configured to use Elastic Load Balancing to distribute traffic evenly across the tasks in a service. Elastic Load Balancing provides two types of load balancers: Application Load Balancers and Classic Load Balancers, and Amazon ECS services can use either type of load balancer.

Continued on following page.

	Kubernetes	Amazon ECS
Auto-scaling for the Application	Auto-scaling using a simple number-of-pods target is defined declaratively with the API exposed by Replication Controllers. CPU-utilization-per-pod target is available as of v1.1 in the Scale subresource. Other targets are on the roadmap.	Amazon ECS can optionally be configured to use Service Auto Scaling to adjust its desired count up or down in response to CloudWatch alarms. Service Auto Scaling is available in all regions that support Amazon ECS. Service Auto Scaling can also be used in conjunction with Auto Scaling for Amazon EC2 instances to scale the cluster, and the service, as a response to demand. Tutorial: Scaling Container Instances with CloudWatch Alarms
Rolling Application Upgrades and Rollback	"Deployment" model supports strategies, but one similar to Mesos is planned for the future. Health checks test for liveness i.e. is app responsive.	Update a running service to change the number of tasks that are maintained by a service or to change the task definition used by the tasks. A updated Docker image of the application can be deployed to the service by creating a new task definition with that image. The service scheduler uses the minimum healthy percent and maximum percent parameters, in the service's deployment configuration, to determine the deployment strategy.
Logging and monitoring	Health checks of two kinds: liveness (is app responsive) and readiness (is app responsive, but busy preparing and not yet able to serve). Logging: Container logs shipped to Elasticsearch/Kibana (ELK) service deployed in cluster. Resource usage monitoring: Heapster/Grafana/Influx service deployed in cluster. Logging/monitoring add-ons are part of official project. Sysdig Cloud integration	AWS CloudWatch can be used to store and analyze logs from the task instance and Docker daemon. AWS CloudTrail can be used to record all ECS API calls. The recorded information includes the identity of the API caller, the time of the API call, the source IP address of the API caller, the request parameters, and the response elements returned by Amazon ECS. Amazon ECS provides monitoring capabilities for containers and clusters to report average and aggregate CPU and memory utilization of running tasks as grouped by Task Definition, Service, or Cluster through Amazon CloudWatch. CloudWatch alarms can also send alerts when containers or clusters need to scale up or down.

Continued on following page.

	Kubernetes	Amazon ECS
Storage	Two storage APIs: The first provides abstractions for individual storage backends (e.g. NFS, AWS EBS, Ceph, Flocker). The second provides an abstraction for a storage resource request (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.	Specify data volumes in Amazon ECS task definitions to provide persistent data volumes for use with containers, or to define an empty, nonpersistent data volume and mount it on multiple containers on the same container instance, or to share defined data volumes at different locations on different containers on the same container instance. There's also an option to use Amazon Elastic File System (EFS) to persist data from ECS containers.
Networking	The networking model (https://github.com/kubernetes/community/blob/master/contributors/design-proposals/networking.md) lets any pod can communicate with other pods and with any service. The model requires two networks (one for pods, the other for services). Neither network is assumed to be (or needs to be) reachable from outside the cluster. The most common way of meeting this requirement is to deploy an overlay network on the cluster nodes.	Amazon ECS strongly recommends launching your container instances inside a VPC to gain more control over the network and offers more extensive configuration capabilities. For more information, see Amazon EC2 and Amazon Virtual Private Cloud in the Amazon EC2 User Guide for Linux Instances. The task definition also has parameters for network settings
Service Discovery	Pods discover services using intra-cluster DNS	ECS recently started providing some basic service discovery or you can use Consul. Here is an article that lays out a reference architecture for service discovery.

It is worth repeating that Amazon ECS is designed for, and provides maximum value, when integrated with Other AWS Services such as Elastic Load Balancing, Elastic Block Store, Virtual Private Cloud, IAM, and CloudTrail. This will likely provide a complete solution for running a wide range of containerized applications or services. On the other hand, Kubernetes is not restricted to run on any particular kind of infrastructure or a specific provider. In fact, Kubernetes can also be easily run on AWS EC2.

Summary

The comparisons above will show that Kubernetes is a powerful and more mature framework than the other tools. It allows a true abstraction layer across private and public clouds, across bare metal and virtualized environments. Kubernetes makes it easier to build and run modern cloud-native apps, since it offers native support for features like service discovery, load balancing and application lifecycle management.

As powerful and easy as Kubernetes is, it is still a non-trivial effort to install Kubernetes in a production environment that meets enterprise-readiness requirements. This typically includes single-sign-on, role-based access control, multi-tenancy etc. There's the critical need to have constant and effective monitoring of Kubernetes clusters to make sure it is available and healthy. The DevOps teams will also have to keep the Kubernetes clusters updated with security and other patches, and upgrading it regularly to keep up with new releases. Most application and DevOps teams would rather offload these tasks and just focus on using Kubernetes to develop awesome new applications, using micro-services and all the other goodness of container technology. This is where a managed solution can help.

Platform9 Managed Kubernetes enables a multi-cloud vision by providing a SaaS-managed offering for Kubernetes. The "managed" experience means Platform9 handles all the nitty gritty
details of Kubernetes deployment and configuration, then ongoing monitoring, troubleshooting and upgrades. Software developers can focus on using the Kubernetes APIs to build cloud-native applications and DevOps teams can focus on realizing a multi-cloud strategy for their organization. What's more, enterprises also get all the enterprise-readiness features such as integration with their choice of persistent storage and networking technology, RBAC support, SSO integration, multi-tenancy and isolation.

Platform9's managed offering includes a high SLA. We provide fully automated deployment and configuration, 24/7 health monitoring and alerting, along with zero-touch updates and upgrades. The Kubernetes environment is delivered in an enterprise-ready state that is applicable whether the clusters are deployed on bare-metal servers, in OpenStack environments or across public clouds like AWS, GCE, Azure.

To experience Managed Kubernetes, please check out the Platform9 Sandbox: https://platform9.com/sandbox.