**Container orchestration**

Container orchestration [book] automates the deployment, management, scaling, and networking of containers. Container orchestration can be used in any environment that runs containers, including on-premise servers and public cloud or private cloud environments, and it’s commonly used when one needs to deploy and manage hundreds or thousands [containers](https://www.redhat.com/en/topics/containers/whats-a-linux-container).

M[icroservices](https://www.redhat.com/en/topics/microservices/what-are-microservices) [book] in containers make it easier to orchestrate services, including storage, networking, and security. Containers give your microservice-based apps an ideal application deployment unit and self-contained execution environment. They make it possible to run multiple parts of an app independently in microservices, on the same hardware, with much greater control over individual pieces and life cycles.

Container orchestration to automate and manage tasks such as:

* Provisioning and deployment
* Configuration and scheduling
* Resource allocation
* Container availability
* Scaling or removing containers based on balancing workloads across your infrastructure
* Load balancing and traffic routing
* Monitoring container health
* Configuring applications based on the container in which they will run
* Keeping interactions between containers secure

Container orchestration tools provide a framework for managing containers and microservices architecture at scale. It allows to describing the configuration of an application trough an configuration file that tells the configuration management tool where to find the container images, how to establish a network, and where to store logs.

When deploying a new container, the container management tool automatically schedules the deployment to a cluster and finds the right host, taking into account any defined requirements or restrictions. The orchestration tool then manages the container’s lifecycle based on the specifications that were determined in the compose file.

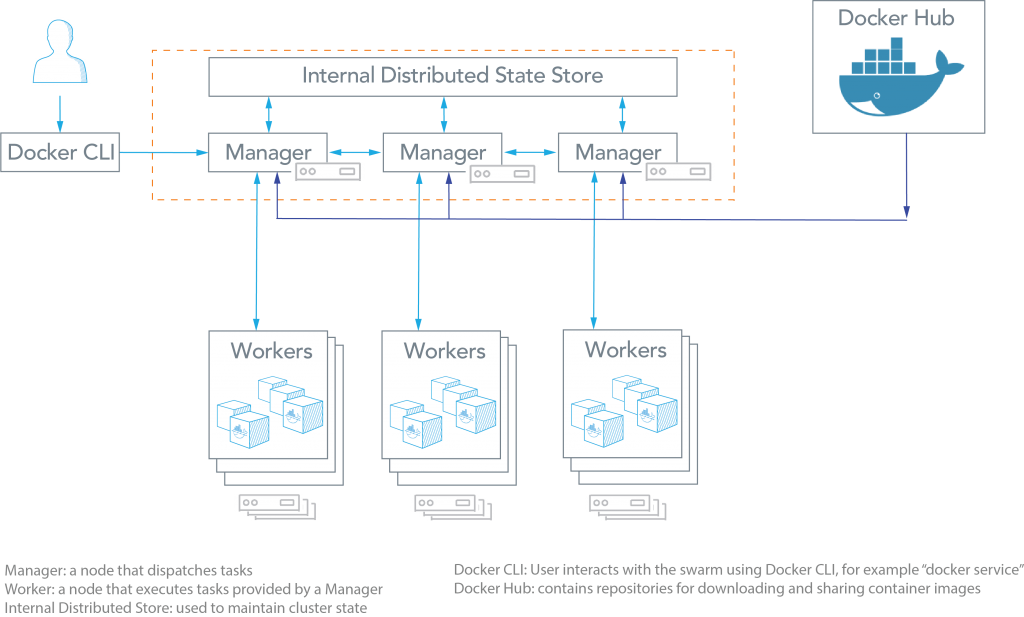
**Container orchestration tools**

There are many container orchestration tools that can be used for container lifecycle management. Some of the more popular options are Docker Swarm and Kubernetes, which we will address further ahead.

Docker Swarm

Docker Engine v1.12.0 and later allow developers to deploy containers in Swarm [Link] 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.



As can be seen from the figure above

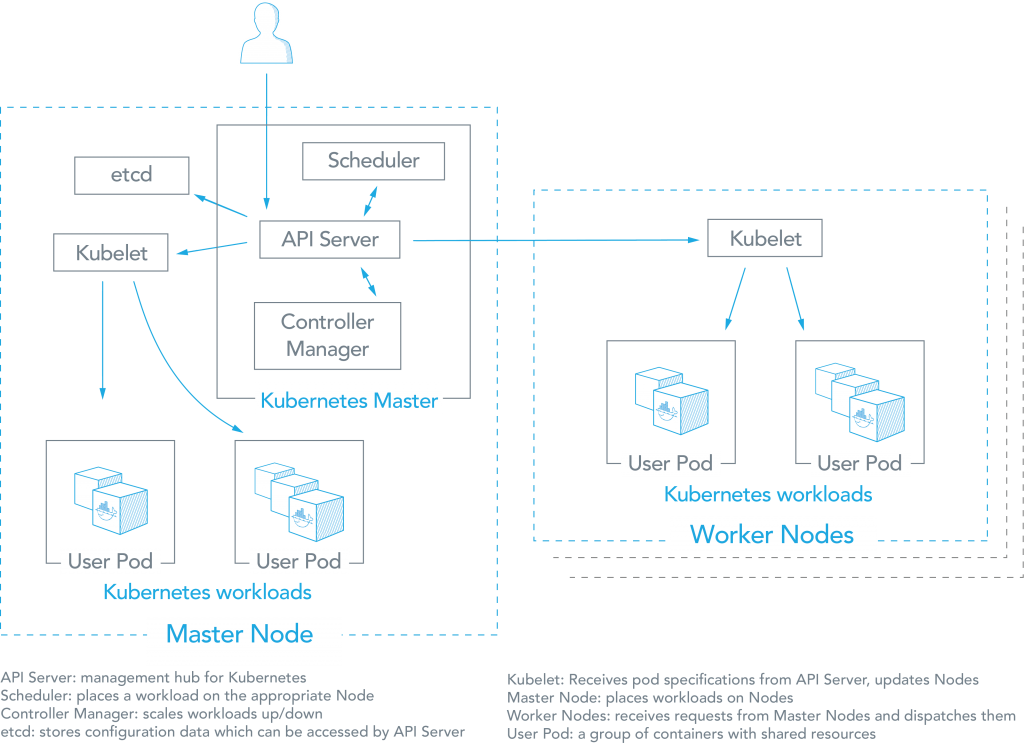
, the Docker Swarm architecture consists of managers and workers. The user can declaratively specify the desired state of various services to run in the Swarm cluster using configuration files.

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, services are orchestrated, 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.

Kubernets

[Kubernetes [Link]](https://kubernetes.io/) 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.



As can be seen 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 the user’s 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.

**Kubernetes vs Docker comparison**

Docker Swarm focuses on ease of use with integration with Docker core components, providing a simple solution that is fast to get started, while Kubernetes remains open and modular, aiming to support higher demands with higher complexity. For much of the same reasons, Docker has been popular among developers who prefer simplicity and fast deployments. At the same time, Kubernetes is used in production environments by many high-profile internet companies running popular services.

Application Definition

Docker Swarm:

Applications can be deployed as services in a Swarm cluster. Multi-container applications can specify 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.

Kubernetes:

Applications can be deployed using a combination of pods, deployments, and services. 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.

Application Scalability Constructs

Docker Swarm:

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. Tasks can be scaled up or down and deployed in parallel or in sequence.

Kubernetes:

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 and pods can be used to run vertically integrated application stacks such.

High Availability

Docker Swarm:

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.

Kubernetes:

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 kubelet and clients. etcd can be clustered and API Servers can be replicated.

Load Balancing

Docker Swarm:

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.

Kubernetes:

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.

Auto-scaling for the Application

Docker Swarm:

Not directly available. For each service, it is possible to declare the number of tasks to be run. Manually scale up or down can be done and the Swarm manager automatically adapts by adding or removing tasks.

Kubernetes:

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.

~~Rolling Application Upgrades and Rollback~~

~~Docker Swarm:~~

~~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.~~

~~Kubernetes:~~

~~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.~~

~~Health Checks~~

~~Docker Swarm:~~

[~~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.~~

~~Kubernetes:~~

[~~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.~~

~~Storage~~

~~Docker Swarm:~~

~~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.~~

~~Kubernetes:~~

~~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.~~

Networking

Docker Swarm:

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.

Kubernetes:

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.

Service Discovery

Docker Swarm:

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. It is possible also to provide a node list as an option at the start of Swarm.

Kubernetes:

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.

~~Performance and scalability~~

~~Docker Swarm:~~

~~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.~~

~~Kubernetes:~~

~~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.~~

|  |  |
| --- | --- |
| Kubernetes2-e1498075079188.pngdocker-swarm-logo-2-219x300.png  **Docker Swarm vs Kubernetes** | |
| **Pros** | |
| * 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. | * Based on extensive experience running Linux containers at Google. * Easy service organization with pods * Can overcome constraints of Docker and Docker API. * Autoscaling based on factors such as CPU utilization. |
| **Cons** | |
| * Does not have as much experience with production deployments at scale. * Limited to the Docker API’s capabilities. * Services can be scaled manually. | * Do-it-yourself installation can be complex, but flexible. * Uses a separate set of tools for management, including kubelet CLI. |
| **Common features** | |
| * Open source projects. * Various storage options. * Networking features such as load balancing and DNS. * Logging and Monitoring add-ons. | |

Cloud Run

In our planning we always aimed to deploy the IS E-Learning platform in a cloud environment as a way to simulate its viability in the future as an online e-learning application that is accessed by multiple users in simultaneous. That said, the details of configuration a container orchestration environment, although interesting, it is really not a subject that we will address in our project, only its practical usage.

Like discussed, Docker Swarm and Kubernetes are great choices for containers orchestration platforms, offering advanced scalability and configuration flexibility, with the possibility to control over every aspect of container orchestration, from networking, to storage. However, at the current state of our project, we don’t need that level of cluster configuration and monitoring, we just want to know how our platform behaves in a cloud environment scenario, and for that reason we choose to deploy our platform on Google Cloud Run.

Cloud Run is a [serverless platform](https://en.wikipedia.org/wiki/Serverless_computing) for stateless containerized microservices, based on containers, that don’t require most known container orchestration features like namespaces, or node allocation and management. It is possible to have all its configurations being fully managed by the Google Cloud Services “black box” - meaning that after implementing a microservice, it will have automatically scalable serverless execution, scaling to zero if there no requests, using no resources in such cases. As managed compute platform, managed Cloud Run also supports essential configuration settings: the maximum concurrent requests a single container receives, the memory size to be allocated to the container as well as request timeout can be configured. No additional configurations or management operations are required.

To deploy *IS E-Learning* platform on Cloud Run, all it is necessary to implement an address management service, by creating one containerized service for each operation - this is creating a Docker Image, which is the Cloud Run’s unit of deployment, for each service: API and execution environments. Once the images have been created and registered in a container registry, they can be deployed to the managed Cloud Run with a single command, making the serviceup and running on a completely serverless platform.