

Apache Mesos

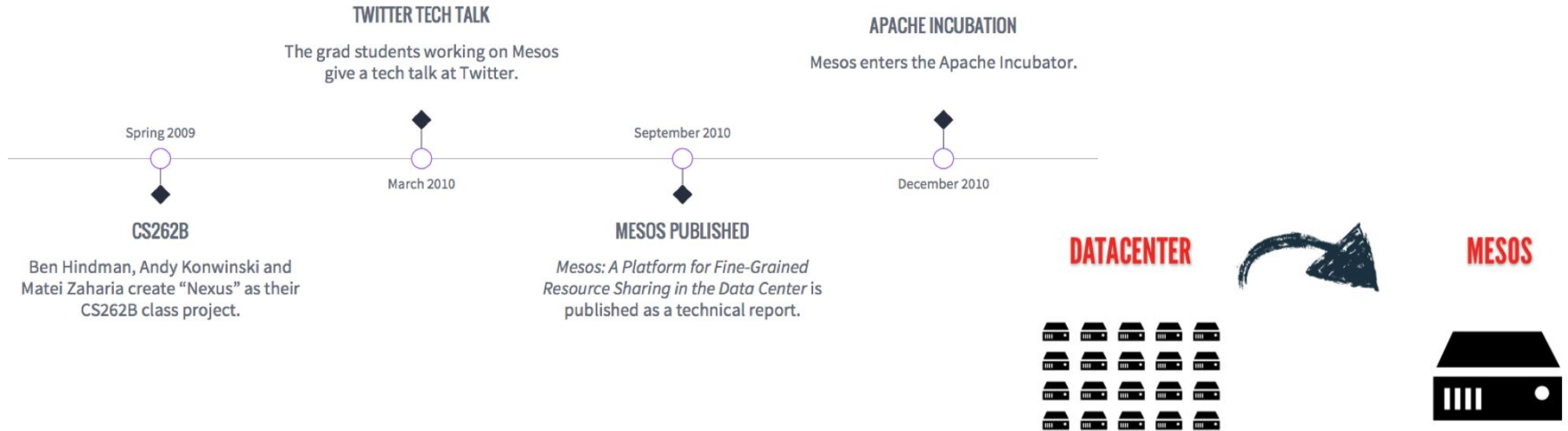
Basic concepts

Marica Antonacci (INFN Bari)
marica.antonacci@ba.infn.it



- What is Mesos
 - Architecture
 - Main components overview
- Two-level scheduling and Resource isolation
- Running long-running services on a Mesos cluster with Marathon
- Executing jobs on a Mesos cluster with Chronos
- Use cases

The birth of Mesos



"We wanted people to be able to program for the datacenter just like they program for their laptop"

Benjamin Hindman, Apache Mesos PMC Chair

<https://people.eecs.berkeley.edu/~alig/papers/mesos.pdf>

Production-proven Web Scale Cluster Managers

Borg/Omega

Tupperware/Bistro

Apache Mesos

~2001

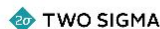
~2007

2010+

Proprietary

Proprietary

Open Source (Apache License)



- Built at UC Berkeley AMPLab by **Ben Hindman** (Mesosphere Co-founder)
- Built in collaboration with Google to overcome some Borg Challenges
- Production proven at scale +80K hosts @ Twitter

© 2016 Mesosphere, Inc. All Rights Reserved.

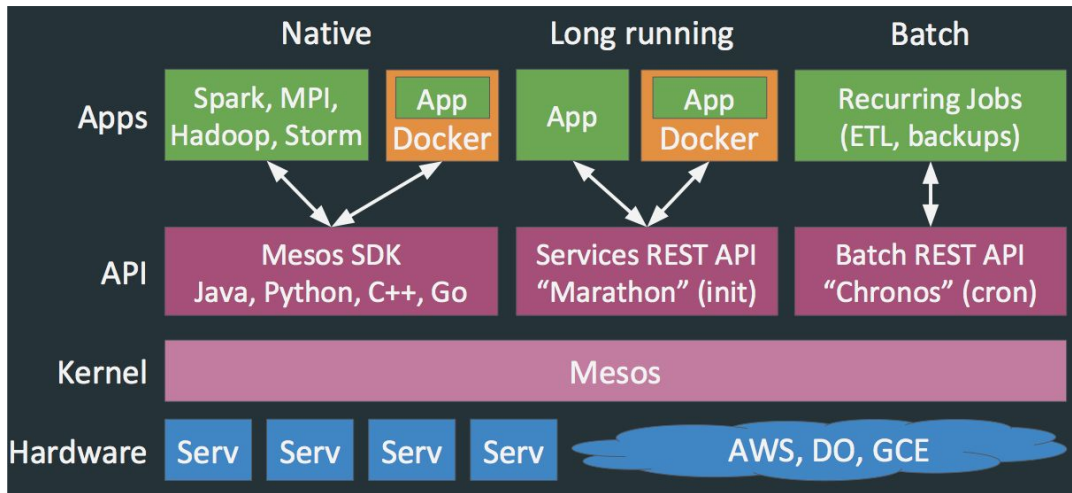
What is Mesos

Mesos has been described as a **Datacenter Kernel** as it provides a single unified view of node resources to software frameworks that wish to consume them via APIs.

Mesos performs the role of an intelligent global level scheduler that can match a massive pool of hardware resources to distributed applications that want to consume these resources.

Mesos aggregates all the resources into a large virtual pool using not just virtual machines and containers but primitives such as CPU, I/O and RAM.

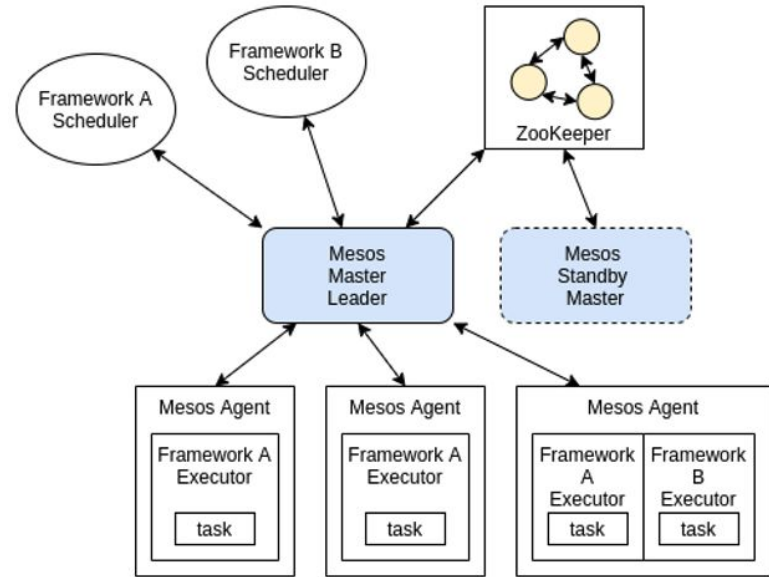
It breaks applications into small units that can be assigned across this pool. Mesos also provides APIs in multiple languages to allow applications to be built for it. Apache Spark, the most popular data processing engine, was built originally as a Mesos framework.



Mesos Architecture

There are 4 important components to run Mesos:

- **Master:** Coordinates the work and decides which framework gets how many resources
- **Zookeeper:** Used as distributed storage, enables the coordination of the masters
- **Slave:** A worker node which provides its resources to run tasks of a framework
- **Framework:** Has a *scheduler* component which decides where a task gets launched and an *executor* which executes one or more tasks at the Slave.

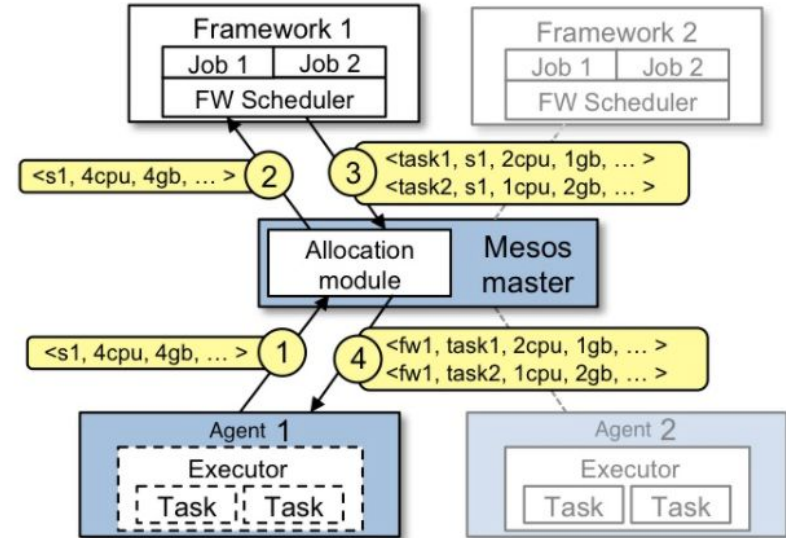


Mesos features

- Fault-tolerant replicated master using ZooKeeper
- Scalability to thousands of nodes
- Isolation between tasks with containers
- Multi-resource scheduling (memory and CPU/GPU aware)
- Java, Python and C++ APIs for developing new parallel applications
- Web UI for viewing cluster state

Two level scheduling

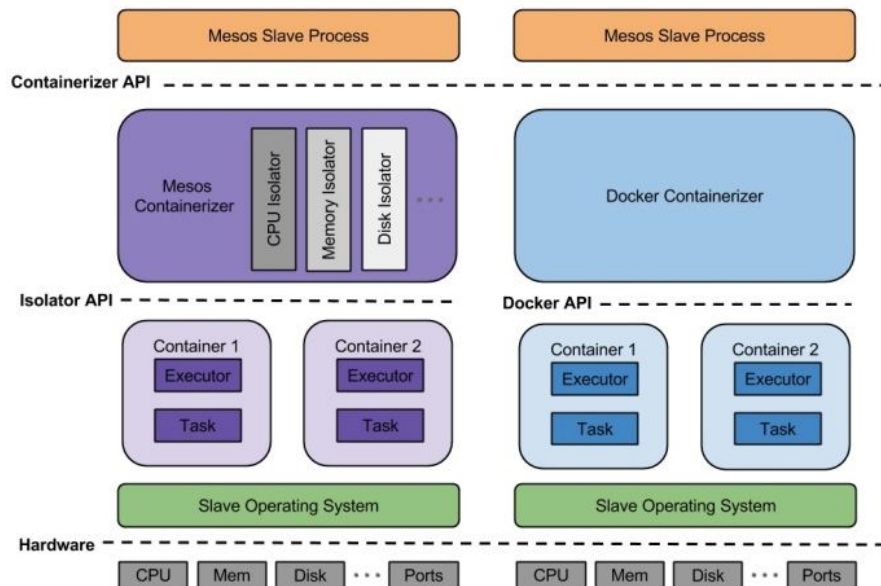
Mesos defines a **minimal interface** that enables **efficient resource sharing** across frameworks and otherwise **push control of task scheduling and execution to the Frameworks**



Resource isolation

The allocation of resources to one framework/job or user should not have any unintended effects on the running jobs.

Mesos provides various **isolation mechanisms** on slaves (*containerizers*) to provide an isolated environment to run an executor and its tasks.



Mesos containerizers

Docker containerizer: This containerizer allows tasks to be run inside docker container

MESOS containerizer: This is the native Mesos containerizer solution. It uses Linux-specific functionalities such as control cgroups and namespaces and allows tasks to be run with an array of pluggable isolators provided by Mesos.

Nvidia GPU support is only available for tasks launched through the **Mesos containerizer** (i.e., no support exists for launching GPU capable tasks through the Docker containerizer).

Note: from version 1.0 on, the Mesos containerizer supports running docker images natively!

Long-running services on Mesos

Marathon Framework

- Marathon is a framework used for running long-running services on Mesos
- Marathon is the equivalent of the **service management system**
 - in Linux, this is commonly referred to as the **init system**.
- Marathon deploys applications as long-running Mesos tasks, both in Linux cgroups and Docker containers.
- It can be considered a **private platform as a service** (PaaS) on which to deploy applications. Marathon does this by launching instances of an application as **long-lived Mesos tasks**

Orchestration with Marathon

1) Configuration/package management

- making sure all the dependencies for a service are met and the environment is configured properly for the service before the service starts



2) Deployment

- Deployment of a service can be complex if service depends on other services and there are constraints about where the service can be deployed



3) Service discovery & Load-balancing

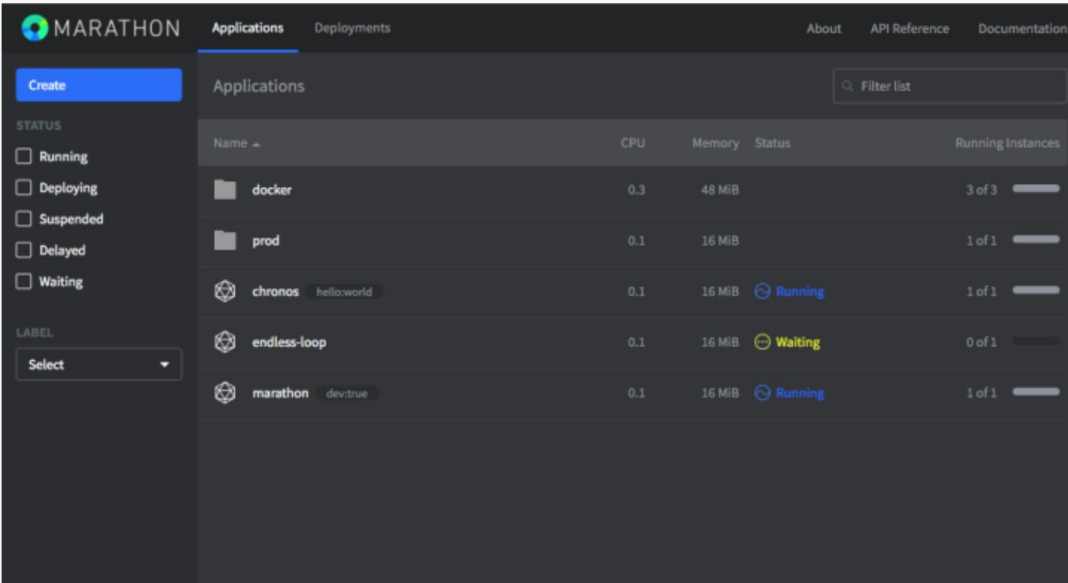
- where are the instances of a particular service running?
- which instance should a given request go to?



Marathon Web Interface

The Marathon web UI provides a convenient interface with Marathon.

Yet it is no longer actively maintained and therefore the usage of the Marathon REST API is strongly recommended to access the latest Marathon features.



The screenshot displays the Marathon web interface. At the top, there is a navigation bar with the Marathon logo, the word "MARATHON", and tabs for "Applications" and "Deployments". On the right side of the navigation bar, there are links for "About", "API Reference", and "Documentation". Below the navigation bar, there is a "Create" button and a search box labeled "Filter list". On the left side, there is a "STATUS" section with checkboxes for "Running", "Deploying", "Suspended", "Delayed", and "Waiting". Below that is a "LABEL" section with a "Select" dropdown menu. The main content area shows a table of applications with columns for "Name", "CPU", "Memory", "Status", and "Running Instances".

Name	CPU	Memory	Status	Running Instances
docker	0.3	48 MiB		3 of 3
prod	0.1	16 MiB		1 of 1
chronos <small>helloworld</small>	0.1	16 MiB	Running	1 of 1
endless-loop	0.1	16 MiB	Waiting	0 of 1
marathon <small>dev: true</small>	0.1	16 MiB	Running	1 of 1

Main endpoints:

API Endpoint	Description
/v2/deployments	Query for all running deployments on a Marathon instance (GET)
/v2/deployments/	Query for information about a specific deployment (GET)
/v2/apps	Query for all applications on a Marathon instance (GET) or create new applications (POST)
/v2/apps/	Query for information about a specific app (GET), update the configuration of an app (PUT), or delete an app (DELETE)
/v2/groups	Query for all application groups on a Marathon instance (GET) or create a new application group (POST)
/v2/groups/	Query for information about a specific application group (GET), update the configuration of an application group (PUT), or delete an application group (DELETE)

<https://mesosphere.github.io/marathon/api-console/index.html>

Running a simple dockerized service

A service is described in JSON format.

The **id** tag is the name of the service. It is displayed in the service list.

The **instances** tag tells Marathon that only one instance is needed. It can be increased or decreased as needed later.

The **cpus** and **mem** tags are hints to Marathon as to what percentage of CPU and the amount of RAM is needed. They do not actually set resource limits in Docker. However, Marathon may kill tasks that use more than the allocated resources. In this case, the application is requesting 25 percent of a CPU and 64 MB of RAM.

The **container** tag is where the Docker container is defined. The **type** tag defines the Containerizer that will be used to run the Mesos task. In this case is set to DOCKER.

The image is set in the **image** tag. This is the same image name that will be passed to docker run.

Finally, the **network** tag is set to BRIDGE, which tells the Docker Engine to use bridge networking and map *containerPort* 80 to the ephemeral *hostPort* assigned dynamically.

```
{
  "id": "simple-nginx",
  "instances": 1,
  "cpus": 0.25,
  "mem": 64,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "nginx:1.11",
      "network": "BRIDGE",
      "portMappings": [
        {
          "containerPort": 80,
          "hostPort": 0
        }
      ]
    }
  }
}
```


Health checks and rolling upgrades

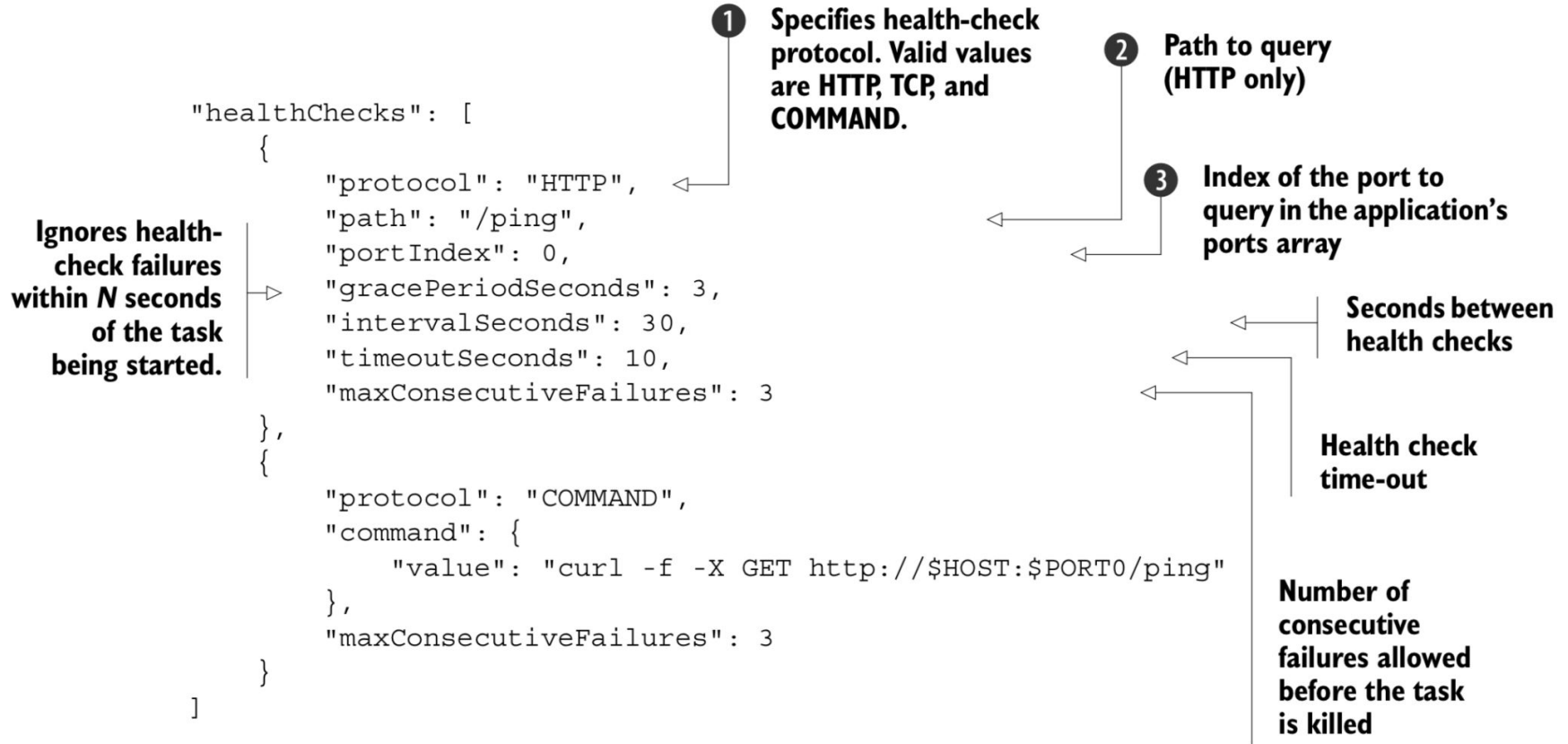
Marathon provides optional **HTTP-** and **TCP-based health checks** for each of the instances of a particular application.

In the event that an instance starts failing its health checks - either by returning an HTTP error code or by failing a TCP connection - the task will be reported as **unhealthy**. After a certain number of failed health checks, Marathon will restart the unhealthy task. The parameters of these health checks are all configurable.

These health checks also allow you to perform **rolling upgrades** of an application, ensuring a minimum level of service, or *capacity*, so that new instances come up healthy before the upgrade proceeds.

Combine these features with dynamically configured load balancers, and Marathon allows for **zero-downtime deployments** of new versions of applications.

Examples of health checks



URIs field & Mesos fetcher

The Mesos fetcher is a way by which **resources** can be **downloaded in the task sandbox** directory while preparing the task execution.

The Mesos fetcher natively supports the **FTP** and **HTTP** protocols, and is also able to copy over files from a filesystem. It also supports all Hadoop client protocols such as Amazon Simple Storage Service (S3), Hadoop distributed Filesystem (HDFS), and so on.

If you specify an **archive** file (for example, zip or tar.gz) in the URIs field, the Mesos fetcher will automatically **extract** the archive for you in the sandbox.

The downloaded URIs can also be **cached** in a specified directory for reuse.

Running stateful application on Marathon

Regardless of the lifespan of the container the data should **always persist**.

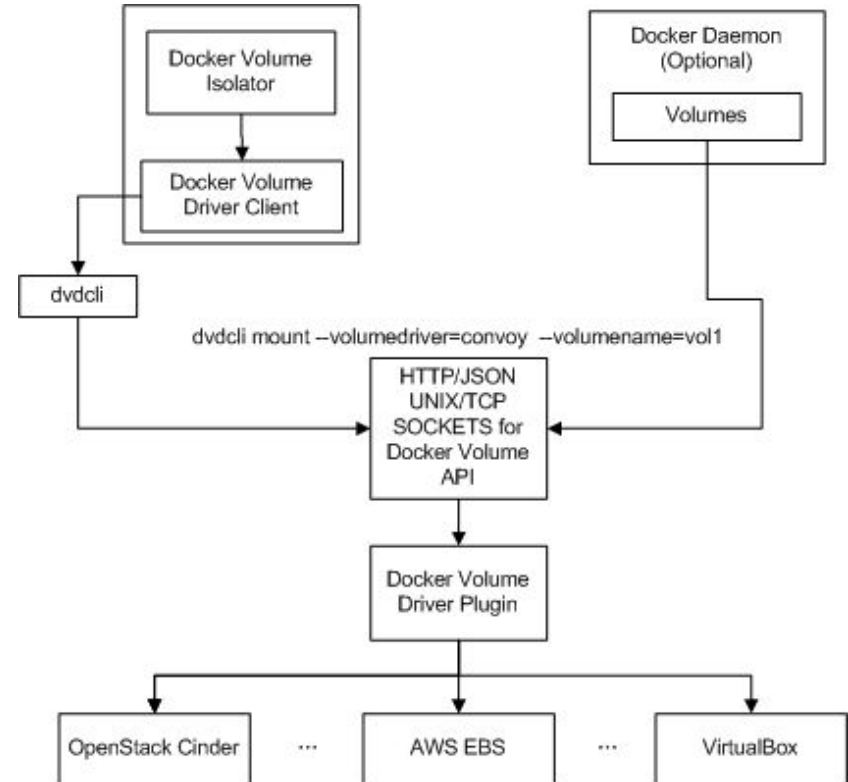
The container could be scheduled to run on any node in the cluster, meaning persistent data may need to be accessed from **any node**.

Marathon supports stateful applications/services by:

- using **local** persistent volumes
 - Failure of a node? data is lost
- using **external** volumes (EBS, Cinder, etc.)

Docker volume driver isolator

The **docker/volume isolator** interacts with Docker volume plugins using **dvdcli**, an open-source command line tool from EMC.



- Provides a vendor agnostic storage orchestration engine
 - Amazon EBS, Ceph, Openstack Cinder, EMC ScaleIO, GCE, XtremIO, etc.

```
"volumes": [  
  {  
    "external": {  
      "name": "mysql-rexray-volume",  
      "provider": "dvti",  
      "options": {  
        "dvti/driver": "rexray"  
      }  
    },  
    "containerPath": "/var/lib/mysql",  
    "mode": "RW"  
  }  
]
```

Marathon networking

Marathon has three networking modes:

- **host**: each application shares the network namespace of the Mesos agent process, typically the host network namespace.
- **container/bridge**: each application should be allocated its own network namespace and IP address; Mesos Container Network Interface (CNI) provides a special *mesos-bridge* that application containers are attached to. When using the **Docker containerizer**, this translates to the Docker “**default bridge**” network.
- **container**: each application should be allocated its own network namespace and IP address; Mesos network isolators are responsible for providing backend support for this. When using the **Docker containerizer**, this translates to a **Docker “user” network**.

Networking mode “host” - example

In this example, we have a single port definition labelled ‘http’ and it’s set to a value of 0, meaning Marathon will choose it on our behalf.

The application takes an environment variable that’s set for us by Marathon called PORT_HTTP, as named under the portDefinitions section. This is passed to the small Golang application and tells it to listen on the value specified in that environment variable

```
{
  "id": "httpi",
  "networks": [ { "mode": "host" } ],
  "portDefinitions": [
    { "port": 0, "name": "http" } ],
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "dcoslabs/httpi:latest",
      "forcePullImage": true
    }
  },
  "instances": 1,
  "cpus": 0.1,
  "mem": 32
}
```


Networking mode “container/bridge” - example

In this example, the service inside the container is running on port 6379 and a pseudo random port on the host will be setup enabling bridge/NAT communication to the container port.

```
{
  "id": "db",
  "instances": 1,
  "cpus": 0.1,
  "mem": 128.0,
  "disk": 0.0,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "redis:3.0.3",
      "network": "BRIDGE",
      "portMappings": [
        { "containerPort": 6379,
          "hostPort": 0, "protocol": "tcp" }
      ]
    }
  }
}
```

Networking mode “container” - example

The “container” mode is the most advanced scenario: a dedicated IP address is allocated to each container.

Containers get their own Linux networking namespace (and thus a dedicated network stack), and connectivity is managed by the underlying software-defined networking (SDN) provider technology.

For example, Calico provides 3rd-party CNI plugin that works out-of-the-box with Mesos CNI.

Calico takes a pure Layer-3 approach to networking, allocating a unique, routable IP address to each Mesos task.

```
{
  "id": "/calico-docker",
  "instances": 1,
  "container": {
    "type": "DOCKER",
    "volumes": [],
    "docker": {
      "image": "mesosphere/id-server:2.1.0"
    },
    "portMappings": []
  },
  "cpus": 0.1,
  "mem": 128,
  "requirePorts": false,
  "networks": [
    {
      "mode": "container",
      "name": "calico"
    }
  ],
  "healthChecks": [],
  "fetch": [],
  "constraints": []
}
```

Service Discovery with Mesos-DNS

Mesos-DNS is a **stateless service** that allows services running in Mesos to find each other through DNS.

It periodically queries the Mesos master(s), retrieves the state of all running tasks from all running frameworks, and **generates DNS records** for these tasks:

- *DNS A records* associate a host name to IPs
- *DNS SRV records* associate services to IPs and Ports

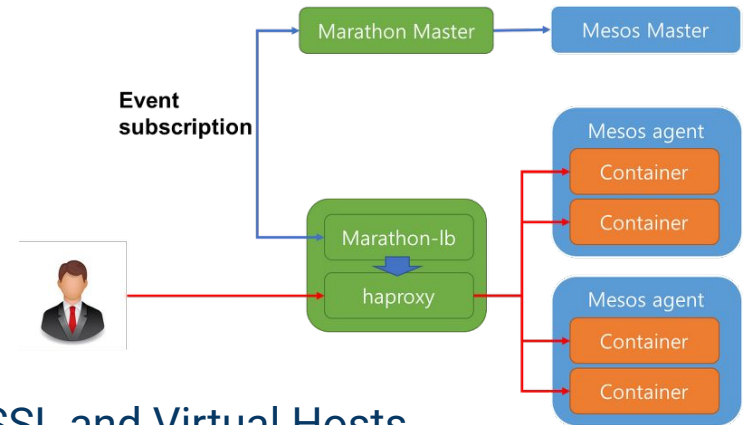
Applications and services running on Mesos slaves can discover the IP addresses and ports of other applications they depend upon by issuing DNS lookup requests or by issuing HTTP request through a REST API.

Service Load-Balancing with Marathon-LB

Marathon load balancer (Marathon-LB) is a proxy server and load balancer for TCP, HTTP, and HTTPS requests based on **HAProxy** open-source software.

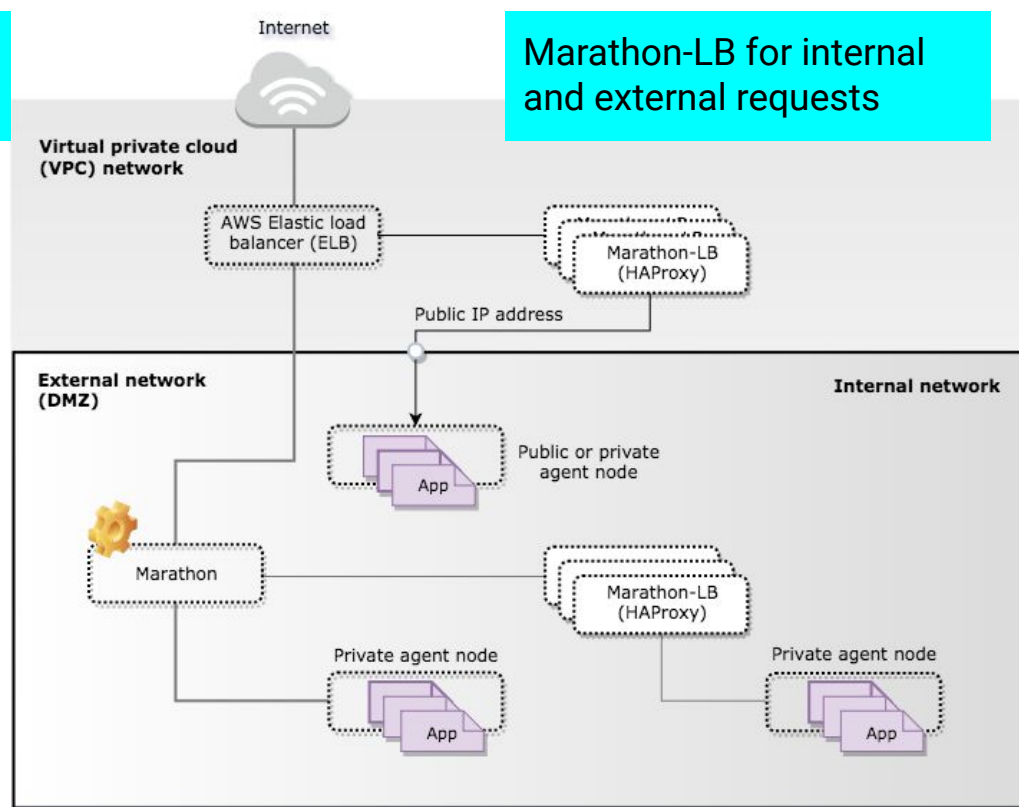
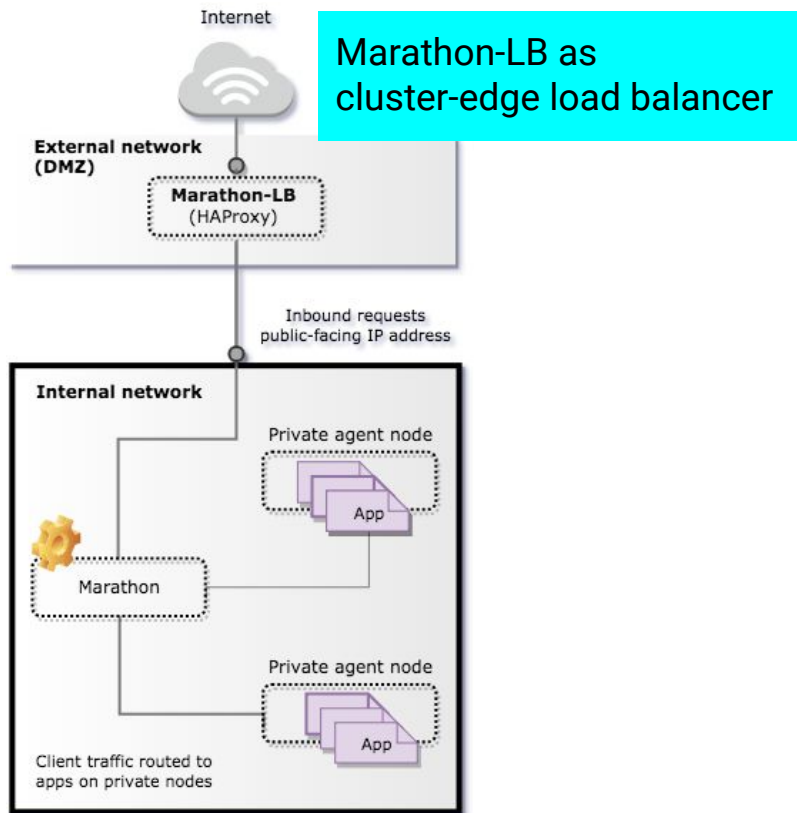
Marathon-lb subscribes to Marathon's event bus and updates the HAProxy configuration in real time.

- Services are exposed on their **service** port as defined in their Marathon definition.
- Apps are only exposed on LBs which have the same LB tag (or group) as defined in the Marathon app's labels (using `HAPROXY_GROUP`). HAProxy parameters can be tuned by specifying labels in your app.



Furthermore, Marathon-lb provides support for TLS/SSL and Virtual Hosts

Marathon-LB topologies



Executing jobs on Mesos

Chronos Framework

Chronos can be considered as a **time-based job scheduler**, such as *cron* in the typical Unix environment.

Chronos is **distributed** and fully **fault-tolerant**, and it runs on top of Apache Mesos.

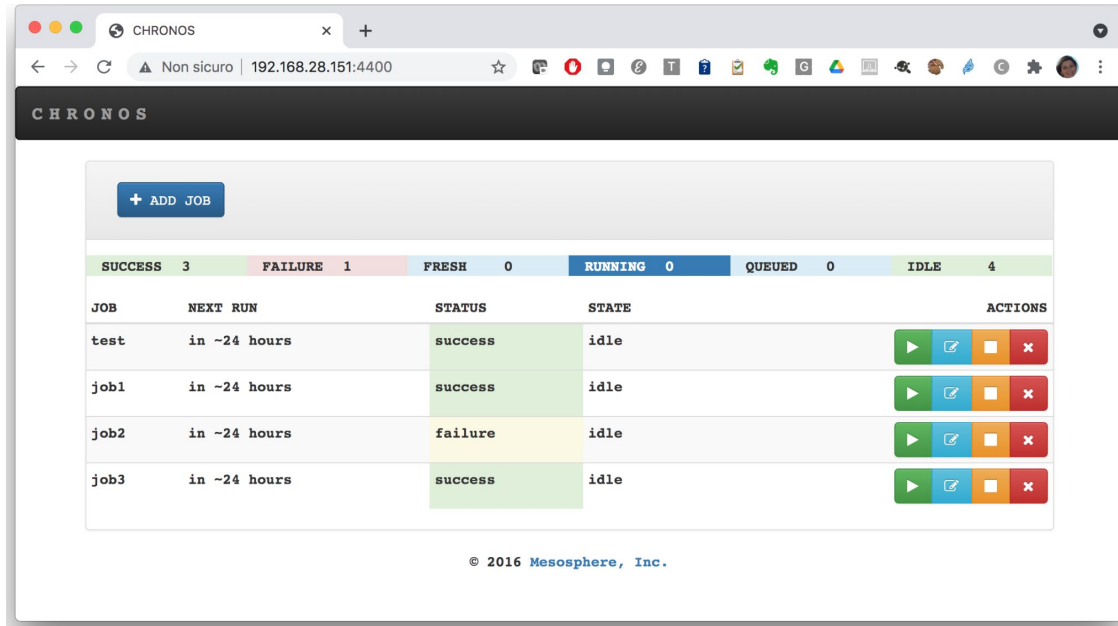
It was originally developed at Airbnb to handle its complex data analysis pipelines

Chronos allows you to run **shell scripts** and is also natively able to schedule jobs that run inside **Docker containers**.

Moreover it supports dependencies and retries.

Chronos web UI

















Chronos comes with a Web UI in which you can see the job status, statistics of the job's history, job configurations, and retries.



The screenshot shows the Chronos web interface in a browser window. The address bar shows the URL 192.168.28.151:4400. The page title is "CHRONOS". There is a "+ ADD JOB" button at the top left. Below it is a summary bar with the following statistics:

SUCCESS	3	FAILURE	1	FRESH	0	RUNNING	0	QUEUED	0	IDLE	4
---------	---	---------	---	-------	---	---------	---	--------	---	------	---

Below the summary bar is a table with the following columns: JOB, NEXT RUN, STATUS, STATE, and ACTIONS.

JOB	NEXT RUN	STATUS	STATE	ACTIONS
test	in ~24 hours	success	idle	   
job1	in ~24 hours	success	idle	   
job2	in ~24 hours	failure	idle	   
job3	in ~24 hours	success	idle	   

At the bottom of the page, there is a copyright notice: © 2016 Mesosphere, Inc.

API Endpoint	Description
GET /v1/scheduler/jobs	This lists all jobs. The result is that JSON contains executor, invocationCount, and schedule/parents
DELETE /v1/scheduler/jobs	This deletes all jobs
DELETE /v1/scheduler/task/kill/jobName	This deletes tasks for a given job
DELETE /v1/scheduler/job/jobName	This deletes a particular job based on jobName
PUT /v1/scheduler/job/jobName	This manually starts a job
POST /v1/scheduler/iso8601	This adds a new job. The JSON passed should contain all the information about the job
POST /v1/scheduler/dependency	This adds a dependent job. It takes the same JSON format as a scheduled job. However, instead of the <code>schedule</code> field, it accepts a <code>parents</code> field.
GET /v1/scheduler/graph/dot	This returns the dependency graph in the form of a dot file

<https://mesos.github.io/chronos/docs/api.html>

Chronos simple job definition

```
{
  "schedule": "R/2021-06-15T22:57:59Z/PT24H",
  "name": "sleep-job",
  "description": "Sleep for 60 seconds and return.",
  "cpus": 0.5,
  "mem": 256,
  "disk": 500,
  "command": "sleep 60",
  "retries": 2
}
```

POST /v1/scheduler/iso8601

```
{
  "schedule": "R/2021-06-15T17:22:00Z/PT2M",
  "name": "dockerjob",
  "container": {
    "type": "DOCKER",
    "image": "ubuntu:latest",
    "network": "BRIDGE",
    "volumes": [
      {
        "containerPath": "/var/log/",
        "hostPath": "/logs/",
        "mode": "RW"
      }
    ]
  },
  "cpus": "0.5",
  "mem": "512",
  "command": "while sleep 10; do date =u %T; done"
}
```

Date and time to start the job,
following the ISO 8601 standard

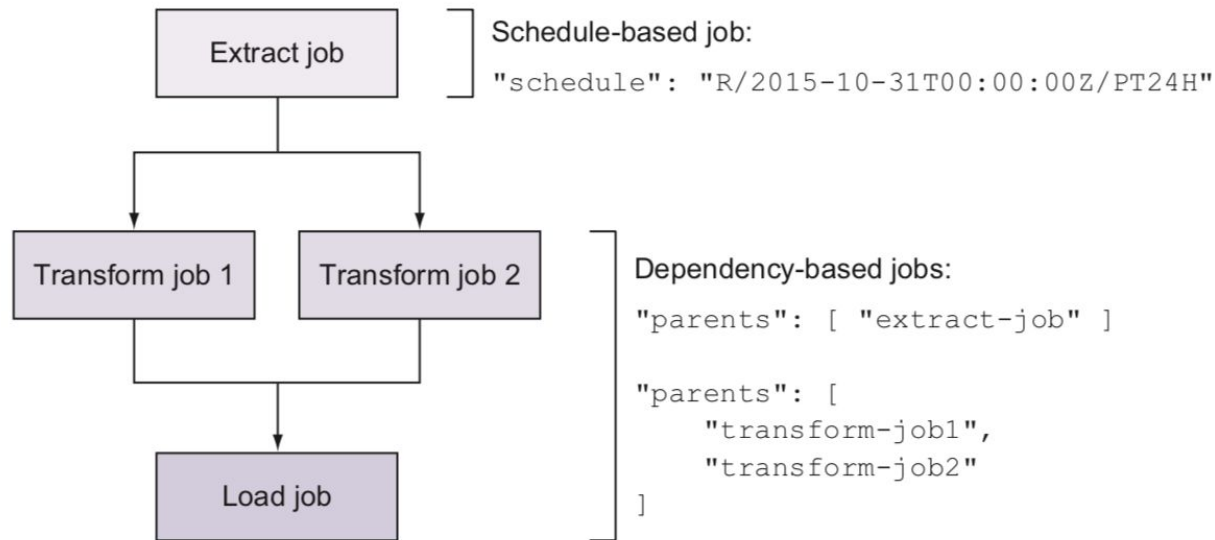
R/2015-10-05T22:00:00Z/PT10M

Number of times to repeat a job.
"R" alone repeats the job forever;
"Rn" repeats the job n times

Run interval, following the
"durations" component of
the ISO 8601 standard

Dependency-based jobs

In Chronos, dependency-based jobs don't contain a schedule field, but instead specify one or more parent jobs (using the parents field) that must complete before that job will run.



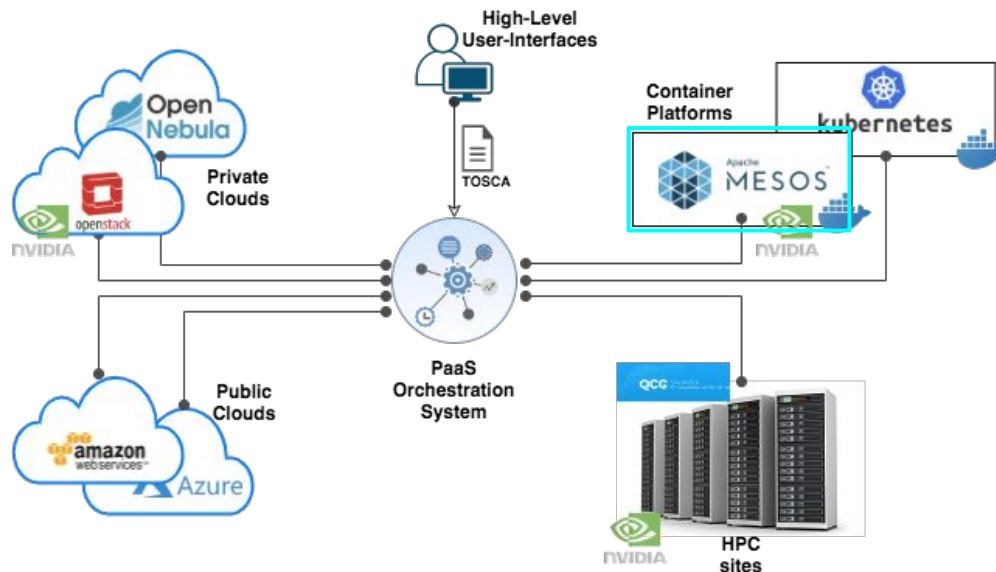
[POST /v1/scheduler/dependency](#)

Mesos Use-cases

- **Mantis**: a reactive stream processing platform. Netflix created this project for its engineering teams to get access to real-time events and build applications on top of them.
 - Mantis covers varied use cases including real-time dashboarding, alerting, anomaly detection, metric generation, and ad-hoc interactive exploration of streaming data
- **Titus**: a Docker container job management and execution platform.
 - Titus uses a master to assign resources from Mesos agents. Titus provides integration into the Netflix microservices and AWS ecosystem, including integrations for service discovery, software-based load balancing, monitoring, and Netflix's CI/CD pipeline, Spinnaker.
- **Meson**: a general-purpose workflow orchestration and scheduling framework that Netflix built to manage machine learning pipelines.
- **Fenzo**: a scheduler Java library for Apache Mesos frameworks that supports plugins for scheduling optimizations and facilitates cluster autoscaling.

<https://netflixtechblog.com/distributed-resource-scheduling-with-apache-mesos-32bd9eb4ca38>

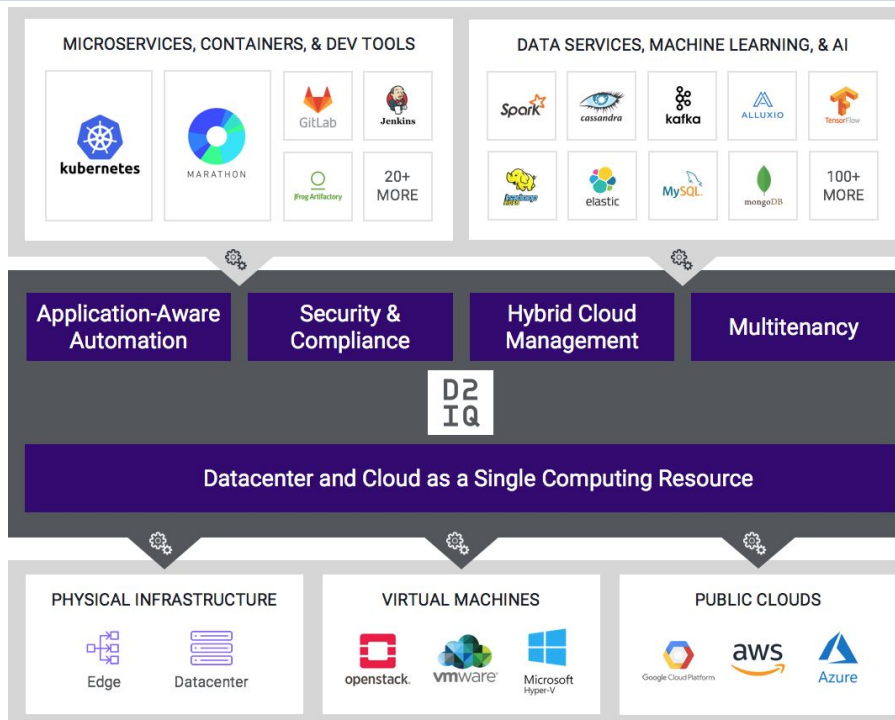
The INDIGO PaaS is able to perform **TOSCA** deployment on Mesos clusters.



The PaaS Orchestrator interacts via REST API with

- Marathon to deploy, monitor and manage Long-Running services, ensuring that they are always up and running.
- Chronos to run user applications (jobs), taking care of fetching input data, handling dependencies among jobs, rescheduling failed jobs.

Mesosphere DC/OS



From <https://d2iq.com/products/dcos>:

End-of-life date for DC/OS: October 31, 2021

It will be replaced by the D2iQ Kubernetes Platform (DKP).

<<Why? Kubernetes has now achieved a level of capability that only DC/OS could formerly provide and is now evolving and improving far faster (as is true of its supporting ecosystem)>>

Mesos - Hands-on

<https://maricaantonacci.github.io/mesos-tutorial/>

References & credits

<http://mesos.apache.org/>

<https://people.eecs.berkeley.edu/~alig/papers/mesos.pdf>

<https://mesosphere.github.io/marathon/>

<https://mesos.github.io/chronos/>

<https://dcos.io/>