



SDP Memo 051: Cloud Native Applications on the SDP Architecture

Document number.....SDP Memo 051
Document Type.....MEMO
Revision.....01
Author..... P.Harding
Release Date.....2018-08-22
Document Classification..... Unrestricted

Lead Author	Designation	Affiliation
P.Harding	SDP Team	NZA, Catalyst IT.
Signature & Date:		

1. SDP Memo Disclaimer

The SDP memos are designed to allow the quick recording of investigations and research done by members of the SDP consortium. They are also designed to raise questions about parts of the SDP design or SDP process. The contents of a memo may be the opinion of the author, not the whole of the SDP.

2. Table of Contents

- [1. SDP Memo Disclaimer](#)
- [2. Table of Contents](#)
- [3. List of Figures](#)
- [4. List of Tables](#)
- [5. List of Abbreviations](#)
- [6. Introduction](#)
- [7. References](#)
 - [7.1. Reference Documents](#)
- [8. Executive Summary](#)
 - [8.1. SDP Use Case and Requirements](#)
 - [8.2. Cloud Native for the SDP](#)
 - [8.3. Cloud Native](#)
 - [8.4. Evaluating Options](#)
 - [8.5. Kubernetes - A Unifying Abstraction Layer](#)
 - [8.6. Software Evolution](#)
 - [8.7. Support and Adoption](#)
 - [8.8. Recommendation](#)
- [9. Understanding Containerisation](#)
 - [9.1. SDP Pipeline Processing Characterisation](#)
 - [9.2. Anatomy of a container](#)
 - [9.3. Container Based Applications](#)
 - [9.4. Container Security](#)
 - [9.5. Container Engine Evaluation](#)
- [10. Value of a Container driven pipeline in HPC](#)
 - [10.1. Selecting the point of Integration](#)
 - [10.2. To Orchestrate or not to Orchestrate?](#)
 - [10.3. Common Requirements](#)
 - [10.3.1. Integrated Software Delivery Life Cycle](#)
- [11. Orchestration Options](#)
 - [11.1. Evaluation Matrix](#)
 - [11.2. Evaluation](#)
- [12. Kubernetes](#)
 - [12.1. Kubernetes Architecture](#)
 - [12.2. Kubernetes Primitives](#)
 - [12.3. Deployment Patterns](#)
 - [12.4. Extending Kubernetes](#)
 - [12.4.1. Dynamic Admission Control](#)

- [12.4.2. Custom Resource Definitions](#)
 - [12.4.3. Custom Resource Definitions & Controllers = Operators](#)
 - [12.4.4. Custom Devices](#)
 - [12.4.5. Storage Plugins](#)
 - [12.4.6. Network](#)
 - [12.5. Kubernetes in HPC](#)
- [13. Resource Management](#)
- [14. Service Discovery](#)
- [15. Scheduling](#)
 - [15.1. The Scheduling Process](#)
- [16. Kubernetes: Cloud Native Implementation Patterns for the SDP](#)
 - [16.1. Accessibility](#)
 - [16.2. Flexibility Satisfies Complexity](#)
 - [16.3. Execution Framework Agnostic](#)
 - [16.4. Kubernetes Containerisation and SDLC](#)
 - [16.5. Applying the Patterns to the SDP](#)
 - [16.6. Kubernetes Overheads](#)
- [17. In Summary](#)

3. List of Figures

- Figure 1 Platform Services C&C view, primary representation
- Figure 2 Example of the anatomy of a container
- Figure 3 The Software Development Life Cycle
- Figure 4 The high level Kubernetes architecture
- Figure 5 Extension points for Kubernetes
- Figure 6 Kubernetes scheduling process flow
- Figure 7 Kubernetes integrated with infrastructure
- Figure 8 Anatomy of a Pod
- Figure 9 Automated DevOps
- Figure 10 GitLab flexible CI/CD pipelines

4. List of Tables

- Table 1 Clear Containers vs Singularity vs Docker
- Table 2 Kubernetes Primitives
- Table 3 Kubernetes deployment patterns
- Table 4 Container Orchestration evaluation matrix

5. List of Abbreviations

AAAI	Authorization, Access, Authentication and Identification
ACL	Access Control List
AKS	Azure Kubernetes Service
API	Application Programming Interface
ARL	Algorithm Reference Library
ASF	Apache Software Foundation
AWS	Amazon Web Services
BDN	Bulk Data Network
CERN	Organisation européenne pour la recherche nucléaire (The European Organization for Nuclear Research)
CI/CD	Continuous Integration/Continuous Delivery
CNI	Common Network Interface
COTS	Common Off The Shelf (Software Components)
CPU	Central Processing Unit
CRD	Custom Resource Definition
CSP	Central Signal Processor
CVFMS	CERN VM File System
DCOS	Data Centre Operating System
DDA	Data Driven Architecture
DevOps	Developer Operations - new breed of systems administration
DNS	Domain Names System

EKS	Elastic Container Service for Kubernetes
FS	File System
GCP	Google Cloud Platform
GSM	Global Sky Model
HA	High Availability
ICD	Interface Control Document
IO	Input/Output
IP	Internet Protocol
IPC	Inter Process Communication
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KVM	Kernel-based Virtual Machine
LSM	Local Sky Model
LTS	Local Telescope State
MPI	Message Passing Interface
NFS	Network File System
NIC	Network Interface Card
NVMeoF	NVMe over Fabrics (Non-Volatile Memory express)
OCI	Open Container Initiative
ORTE	Open Runtime Environment
OS	Operating System

PFS	Parallel File System
QA	Quality Assurance
QEMU	QEMU is a free and open-source emulator that performs hardware virtualization
RDBMS	Relational Database Management System
RSC	Regional Science Centres
SAFe	Scaled Agile Framework
SAP	Systeme, Anwendungen und Produkte in der Datenverarbeitung - ERP: Enterprise Resource Planning software company
SDI	Software Defined Infrastructure
SDLC	Software Development Life-Cycle
SDP	Science Data Processor
SKA	Square Kilometre Array
SRC	SKA Regional Centre
SSD	Solid State Drive
TBD	To be determined
TM	Telescope Manager
URI	Uniform Resource Indicator
VM	Virtual Machine
YAML	Yet Another Markup Language

6. Introduction

Within the SDP Architecture bridging the Execution Control and Platform Services, there is the requirement for the twin capabilities of resource management and scheduling. These capabilities underpin the ability of the SDP to optimise the use of the available platform resources within the allocated budget of power, compute, network and storage. This becomes part of the critical chain of dependencies that will build the successful implementation.

The rise of Cloud Computing Infrastructure has become an opportunity to reimagine the management tools, development frameworks, and platform services integration. These new tools and processes are required to cope with the expanding needs of organising and managing resources at ever increasing scales. This has been borne out of a need to satisfy the varied computational requirements for everything from global online trade, to big data analytics through to HPC in the research sector. Without efficient and responsive tools, the allocation and tracking of resources has become difficult or even impossible leading to under utilisation or over commitment. At best resources remain idle, and at worst critical workloads fail to run.

This opportunity for efficiency has been driven by the advent of Containerisation that has enabled workloads that traditionally operate at the node level to be scheduled at the fraction of a host level, and in most cases removing the need for KVM style virtualisation all together. This has been further extended by systems for Container Orchestration, which now abstract the primitives of Compute, Network and Storage so that a collection of once individual hosts are now addressed as a whole.

These new models for scheduling workloads and the associated resource management have given rise to a new kind of application architecture and associated Software Development Life Cycle which is now referred to as Cloud Native - the delivery and management of services based on these new architectural paradigms enabled by Cloud Computing Infrastructure.

This document aims to catalogue the capabilities offered by Cloud Native platforms and illustrate how an approach based on this solution will benefit the SDP, supporting the architecture, and meet its requirements.

The document is broken into two parts - the first is a comprehensive executive summary that should be sufficient for those that want to get a sense of the proposition, and the rest goes into considerable detail covering:

- What is Containerisation, the relevant concepts to SDP workloads, resource management and scheduling, and containerised workload security
- Cloud Native orchestration solution options
- The Kubernetes architecture and capabilities
- Resource management
- The importance of service discovery
- Scheduling SDP workloads
- Implementation patterns

From these various aspects of a Cloud Native platform architecture, a summary case is made for this as a solution architecture choice for the SDP.

7. References

7.1. Reference Documents

Reference Number	Reference
AD01	Platform Services Component and Connector View
AD02	Operation System Component and Connector View
AD03	Execution Control Component and Connector View
RD01	SKA1 LOW SDP - CSP Interface Control Document - 100-000000-002

8. Executive Summary

8.1. SDP Use Case and Requirements

A central Platform Services requirement for the SDP is the twin capabilities of resource management and scheduling . Resource management entails the identification, accounting and monitoring of platform infrastructure components. Scheduling is the efficient prioritising and packing of tasks from all kinds of workloads against available resources. In order to successfully manage resources and schedule workloads, it is paramount that there is a complete and accurate view of capacity to plan and place tasks against.

For the SDP, all of this efficient use of capacity must happen against a backdrop of:

- Specific power consumption caps
- Near 24x7 operational requirements to meet the widest possible observation schedule
- A mixture of batch and near real-time stream based processing, driven by storage capacity requirements - raw data must be discarded as soon as possible with a buffer of approximately 7 days
- an SDLC strategy that supports distributed design and development, wide variations in languages, tools, resource capacity, and longevity
- Work in collaboration with SRC and other scientific endeavours such as CERN, and where possible provide isolation and encapsulation with a common approach to deployment that abstracts running code from physical infrastructure
- cope with a heterogeneous landscape evolving over time for 50 years
- promote reusable software components that enable efficient development of pipeline solutions and promote reuse beyond the SDP in the SRC
- satisfy the risk mitigation strategy of adopting COTS software, and industry standards and best practice for software development and service delivery

8.2. Cloud Native for the SDP

Cloud Native¹ is a set of industry defined standards that describe a portable abstraction for the management of software services based on containerisation, deployed on cloud based infrastructure.

Cloud based infrastructure provides the primitives of compute, network, and storage resources (hardware) as a commodity either on or off premises, that can be managed via a common API resulting in Infrastructure as a Service.

Cloud Native extends these infrastructure primitives by adding an abstraction layer that turns the software lifecycle management for a service into a common set of units and processes. This shields the architect, developer, and administrator to a large degree from the hardware dependent issues of the platform. This gives a greater degree of portability for all the phases of the SDLC enabling component software to be shared between operational environments with consistency and reliability, promoting reuse and cohesion between all stakeholders. Within the

¹ <https://www.cncf.io/>

SDP this ranges from developers on their desktops, through CI testing and QA environments on to the heterogeneous production environment and out to the SRCs.

This architectural paradigm sits neatly within the existing SDP proposed architecture. In the CDR document “Platform Services Component and Connector View” [AD01], the Orchestration Services component expresses the requirements of a solution that provides resource management and scheduling capabilities as platform level shared services. These capabilities are consumed by Execution Control (Operation System Component and Connector View [AD02], and Execution Control Component and Connector View [AD03]) where the observation schedule communicated by the Telescope Manger (TM) are turned into requests for running software configurations on the platform.

Execution Control is responsible for communicating with the Platform Services Orchestration Services to negotiate resources and scheduling. Platform Services requires a mechanism that tracks resource allocation (Compute, Network, and Storage) and provides a common interface for the allocation and tracking of the consumption of these resources. This will effectively be a state machine that Execution Control can real-time query to understand what is available, and what the current state of the running applications are.

Platform Services can support these requirements by the combined capabilities of Cloud based infrastructure services (OpenStack), and Cloud Native platform solutions.

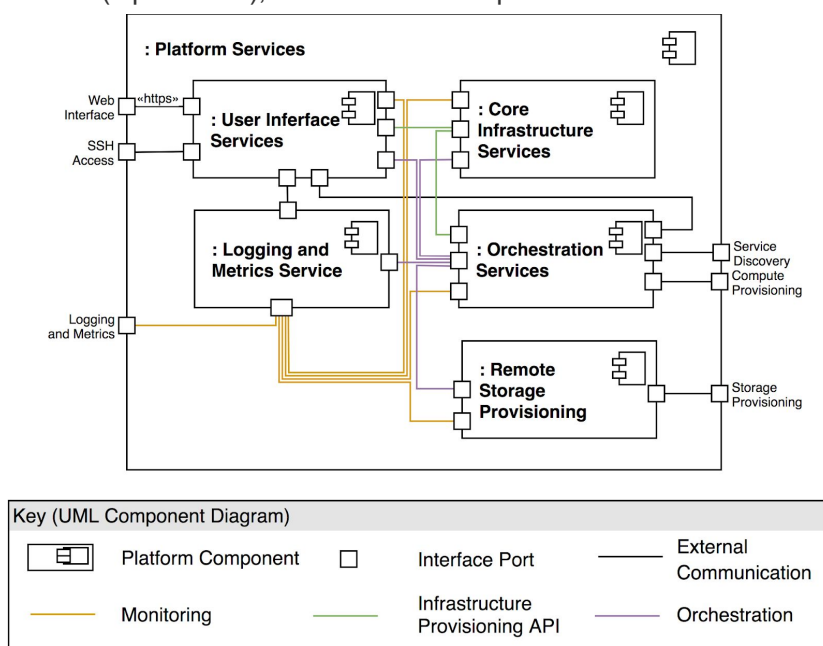


Figure 1 - Platform Services C&C view, primary representation

8.3. Cloud Native

Cloud Native requires a shift in solution architecture in order to realise the full potential. This can be summarised as:

- Adoption of Containerisation is mandatory - without it the encapsulation of components for orchestration is not possible on an economic basis

- The Cloud Native Orchestration layer (Kubernetes, Docker Swarm, DCOS) must integrate with Cloud Platform services to enable resource management, scheduling and auto-scaling
- Developing applications with a micro-services oriented architecture will give the encapsulation, and modularity to take advantage of the independently scalable units. This enables finer grained targeting of resource allocation, leading to the improved efficiency

The pros and cons of choosing this approach are:

Pro	Con
Will provide specific guidance to software development through established design and deployment patterns - standardisation	An opinionated software design pattern - potentially restricting freedom
Orchestration Engine abstracts compute, network and storage resources from development and deployment	The Orchestration solution is a major investment with similar consequences to the Cloud Computing infrastructure solution (ie. OpenStack is foundational)
Highly portable environment including development, test, and production	Platform overheads for the Orchestration solution, and Container Engine
Broadly supported industry initiative over 8 years+	
Broad community support and uptake - high level of documentation, training, skills, tools, and software available	
The modular design, encapsulation, and abstraction provides a strong path for software refresh cycles, migration and dealing with obsolescence	
Will provide a common currency for sharing Intellectual Property with SRC, and other scientific endeavours	

8.4. Evaluating Options

As part of assessing the industry offerings in the cloud native platform space, Docker Swarm, DCOS (Mesosphere) and Kubernetes versions 1.6 and 1.10 have been evaluated with selection based on adoption, support, COTS status, and proven record in production deployment. All solutions have their merits but Kubernetes 1.10 stood out as the best fit for purpose. The most notable characteristics are:

- Provides a platform for defining and managing all data centre workloads (batch, daemon, stateful, real-time, internal/infrastructure services)

- Fundamentally API driven, with carefully constructed primitive components that can be arranged in any number of patterns to make higher order applications providing flexibility and future proofing
- Largest market/mindshare - has achieved the largest cross industry participation and support
- Successfully redefined the fundamentals of software delivery and data centre management providing direction and leadership to the next evolution of cloud based service delivery

8.5. Kubernetes - A Unifying Abstraction Layer

Throughout the investigations behind this memo, Kubernetes demonstrated that it has redefined the landscape for software design, development and deployment as Cloud Native. This has been achieved by adopting containerisation, having a clearly defined architecture that is readily exposed by an API, and cleverly abstracting deployable objects from infrastructure. The most revealing example of this is the framework for developing Operators. Operators are user defined extensions that expose new object types as first class elements in the Kubernetes API. By composing new Object types out of the basic Kubernetes primitives, new first class managed components can be created eg: a PostgreSQL database object, a Tango device object, or a Prometheus monitoring database. These objects can then be created, deleted, managed, and monitored just like any other Kubernetes object.

8.6. Software Evolution

The abstraction layer is pitched at a level that decouples design and development of tasks from the intricacies of compute, network, storage and specialised device support. This in turn frees both infrastructure support and software development from being as tightly coupled as in legacy architectures. An example of the benefits of this is coping with storage evolution. If the future promise of NVMeoF based flash storage became cost effective for the hot buffer in the SDP, then the strategy for dealing with integration and migration for the SDP is easily broken into two parts. The new physical storage facility (flash enclosures, Network adapters etc.) is integrated with Infrastructure Services (eg: OpenStack). Then the new storage capability is integrated with Kubernetes as an alternate storage class implementation with a driver that exposes the storage to Pod Containers. From the Pod (schedulable task) point of view nothing changed other than the storage class name referenced at the time the requisite Persistent Volume Claim is created. In this example, Kubernetes is providing an abstraction layer from Infrastructure Management, so that infrastructure and application evolution are decoupled and able to evolve independently and on different time scales.

Additionally, in the case of storage, Kubernetes manages Persistent Volume Claim reference counts so acts as a registry data usage. storage classes, persistent volumes, and PersistentVolumeClaims can form the basis of the storage management subsystem eg: Buffer Registry, and Buffer Provisioning.

8.7. Support and Adoption

Kubernetes is available on OpenStack (Magnum), AWS (EKS), GCP, Azure (AKS), and OpenShift (RedHat) or can be deployed on bare metal. Through community supported Operator plugins and Helm Charts, Kubernetes can provide key platform services such as Monitoring and Logging, RDBMS, and Data Queues that can be managed as first class objects and stateful services. The community support in this area is rapidly overtaking OpenStack seemingly because the APIs and Developer tools have drastically lowered the barrier to entry for second tier projects. An example of this is OpenStack Trove - RDBMS as a Service, which is losing ground to KubeDB the equivalent in the Kubernetes landscape, which is providing production grade database management.

A key requirement for the success of a platform is accessibility for developers, and integration with the SDLC tool chain. Kubernetes provides Minikube which runs comfortably on the desktop, is strategically aligned with Docker for Container development and management, and has first class integration with GitLab, ElasticStack and Prometheus for automated DevOps, monitoring, logging, accounting and alerting.

Within the last 12 months (since version 1.6) Kubernetes has significantly matured, where it is now providing well curated releases, has stabilised the API, and has developed and carried a broad community of contributors with it. The applied knowledge of Google Engineers learnt through the development of the internal "Borg" precursor project (15 years in the making) is showing through in the depth and sophistication of the design where everything including the internal components use and expose public APIs. Kubernetes is now arguably the Platform as a Service that future higher order services will spring from such as Function as a Service (currently available as an Operator: [Kubeless](#)) and other Serverless Frameworks.

It is also telling that both CERN and SAP have developed projects where OpenStack is deployed on top of Kubernetes. Kubernetes is fulfilling the role of a distributed operating system, and OpenStack is providing the Infrastructure Management services effectively turning the roles of Infrastructure Services and Platform Services inside out. This is aided by OpenStack Containerisation projects such as Kolla and Loci that provide curated releases of core components. Experimentation at CERN has advanced to the point where a new OpenStack region has been deployed with Kubernetes as the foundation and the OpenStack control plane delivered on top.

The open source nature of the Kubernetes project coupled with it's broad and engaged community base gives a high degree of assurance over it's longevity and support. A [timely reminder](#)² of this are the problems experienced in the [Singularity](#)³ and [Shifter](#)⁴ containerisation solutions that have recently been caught with a serious security vulnerability that was identified and plugged by the Docker community more than a year prior. This is an example of openness and widespread community adoption and support at work where 'many eyes over code' makes for better and safer code.

² <https://www.nextplatform.com/2018/05/04/hpc-container-security-fact-myth-rumor-and-kernels/>

³ <https://singularity.lbl.gov/>

⁴ <https://github.com/NERSC/shifter>

8.8. Recommendation

On the basis of the discoveries in this report, the recommendation is that the SDP should include Kubernetes as part of adopting a Cloud Native service delivery approach encompassing:

- Core container orchestration service, backed by the Docker container engine
- Central resource management and scheduling solution required by Execution Control and Platform Shared Services
- The basis of the storage management subsystem eg: Buffer Registry, and Buffer Provisioning
- Operators and Helm templates as the standard implementation for core and common platform services such as RDBMS, monitoring and logging, Tango, Data Queues etc.
- The common currency for exchange of application intellectual property with SRCs and the wider scientific community
- Create a clear distinction between Cloud Infrastructure services managing the deployment of compute, network and storage primitives, which are then brokered to running services and applications through scheduling and resource management layer provided by the container orchestration engine.

There is the additional potential opportunity for using Kubernetes as the minimum viable Execution Framework most notably in the context of batch processing, as it has a fully featured, dependency controlled job scheduler.

9. Understanding Containerisation

A key emerging technology that is changing how software is designed, developed, deployed and managed is Containerisation. It is technology that enables a form of lightweight virtualisation where the unique dependencies of a running application can be packaged into an isolated environment complete with resource management, quotas, accounting and logging. These containers can then be far more densely packed on a physical host than equivalent VM technology, with little or no overheads.

Cloud Native is the movement towards exploiting the benefits of this style of service delivery, where software is architected as a cluster of self-contained components (often described as micro-services) that can be independently managed and scaled. Each component is containerised, and encapsulated in deployment descriptors and processes that aim to be infrastructure independent.

9.1. SDP Pipeline Processing Characterisation

SDP Compute is described as a Data Driven Architecture (DDA). This is because the Compute problem is characterised by extremely high data throughput rates that require a multi-stage rendering process to deliver the desired Science Data Products. In order for the current state of the art compute resources to cope with these volumes, it has been recognised that the computational flow will need to split the data into many channels and packets, iteratively process these packets, and combine the reduced results back into the target Science Data Products. This is analogous to the large scale analytics paradigm of Mapreduce.

An SDP Pipeline component is a unit of work that will take in one or more packets from one or more frequency channels delivered from the CSP matched with other sources of context information (LTS, LSM/GSM, timing, configuration etc), and then perform some kind of computationally expensive process to render an output dataset. These outputs may then be passed on to the next process that may reside on the same compute node, another node within the same compute island, another compute island or the data may need to be stored (permanently) until the next steps are scheduled.

These diverse processing and orchestration scenarios are the range of possibilities but it is highly desirable that the scheduling component of the SDP Pipeline should attempt to place as much of the processing pipeline within the same node or island as possible to optimise throughput.

This leading edge of optimisation will be where adjacent components will be able to directly share resources such as memory, pipes, SSD storage and physical network interfaces.

9.2. Anatomy of a container

A containerised application is a set of (ideally) one or more processes that run on the host OS kernel within its own name-space with specific control (group) structures applied. As a result the

container has many of the same characteristics of a separate virtualised host with a network stack, file-system, and OS profile (Kernel, CPU, Memory).

Containers are derived from images, which are used as the starting point for the launched application. Images are made up of one or more read-only layers which are unified (one on top of the other - the [Union File-system](https://en.wikipedia.org/wiki/UnionFS)⁵) to produce a complete file-system. Different images can share layers as they are identified uniquely using a hashing algorithm. The unified layers are effectively a file-system directory structure often analogous to an entire working OS (libraries, executables, drivers etc.). At run time, a final empty read/write layer is placed on top to hold the IO of the running container.

The life cycle of a container is managed by a container engine (Docker, Rkt, LXC/LXD). The engine configures the interface of the container with the host OS including mounting in file systems, CPU, Memory, Kernel tuning parameters, and the network stack.

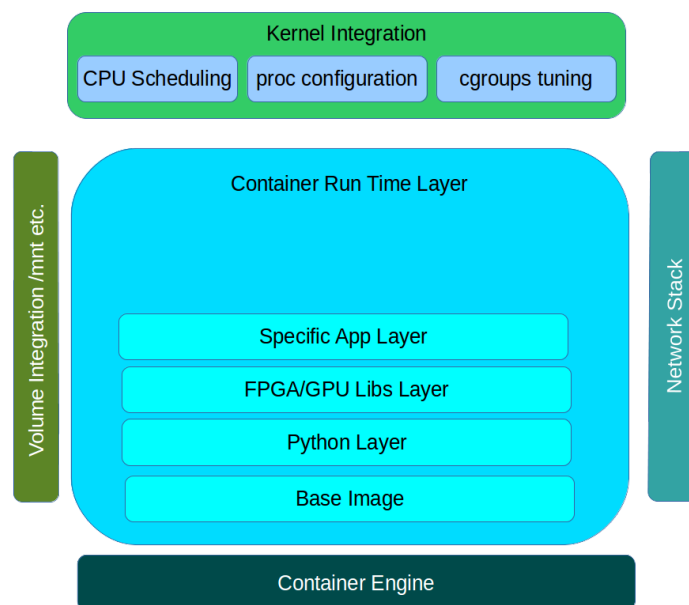


Figure 2 - example of the anatomy of a container

There are many benefits to application containerisation systems:

- the image sizes are dramatically reduced compared to other virtualisation technologies (to as little as a few MB or the size of a single statically linked executable), and lower image layers can be shared between divergent containers
- the start-up times are reduced (there is no OS boot time required - just the process(es) is started)
- and the operational density is increased considerably because of the removal of running a kernel and supporting OS processes per execution unit
- Applications with differing software version dependencies can run on the same host in complete isolation

⁵ <https://en.wikipedia.org/wiki/UnionFS>

- Applications can run on top of divergent hosts, and in heterogeneous environments
- Identical execution environments can be shared through exchange of images - applications take on some of the characteristics of data - they are shipped to the platform for processing

Container based application development is a step change. Because of the lightweight abstraction of containers and their inherent portability, it is possible to look at application development in a new light. For instance portable containers that can utilise generic computing resources can be moved to the data rather than the traditional design philosophy of pulling data to the application. The benefit of this paradigm is in the cost - time, compute, vs storage, vs data transfer - trade off.

9.3. Container Based Applications

The optimal design for a container based application is to have single task containers that are coordinated together to produce the desired service. Each container instance must be expendable i.e. it must not store any non-volatile state within, and if instances are destroyed and restarted then the application does not suffer. This design philosophy maps well to modern systems administration principles of “Cattle not Pets”, where the idea is to compose services out of expendable units that are derived from automated configuration processes so that they can be rebuilt, upgraded, and patched automatically at any time. This kind of approach enables systems operations to be:

- More reliable, and robust
- Continuous
- System migrations are easier
- Configurations are managed like code so conform to modern SDLC practices
- Operational management can scale to 100,000s of units

It is important to not fall into the trap of thinking of containers as virtualised hosts - the litmus test is they should not require a process control system.

Multiple services in a container indicates that there is not adequate separation of concerns. These dependent services cannot be developed, scaled, upgraded and managed independently leading to brittle and constricted platform management.

9.4. Container Security

All container engines are bound by the same set of underlying capabilities defined by the Linux Kernel (at least in the Linux world) that they run on and controlled by specific additional Kernel plugins. The things that divide the Engines is how they make use of these features, and how these features are presented to the end user.

Containerisation solutions all run on the same underlying Kernel features, namely:

- Namespaces
- Cgroups
- Capabilities (or caps)
- File-system solution

- Additional Kernel level security extensions such as AppArmor, GRSEC⁶, SELinux

Namespaces enable the separation of different systems resources between groups of processes. These are typically (but not limited to):

- pid - process isolation
- net - network interfaces
- ipc - inter process communication
- mnt - file-system mount points
- uts - kernel and host identification (eg: hostname, uname)
- user - process id mapping between host and container

The combination of these control points are what makes a container appear isolated or virtualised from the underlying host - the processes inside the container see their own virtualised environment as a result.

Cgroups (control groups - memory, cpu, cpuset, blkio, net_*, devices) are a mechanism for applying resource limits, prioritisation, accounting, device access and control of process groups. Within the context of containers, this enables containers to be given memory and CPU time slice limits, and for containers to be pinned to one or more CPUs.

Capabilities are like a permission system for process groups. Containers can be given (or have taken away) rights such as SETUID (set process userid), SETGID (set process groupid), SETPCAP (set capabilities on a process scoped by parent capability set) and many more.

The file-system solution defines what houses the container directory structure, and this can be anything from a separate file-system image file on the host server, through to a directory tree that is used as the root directory location for the starting container process (similar in effect to chroot at a directory location).

Comparing container Engines ⁷ - the software component that manages and runs containerised software - does not give a complete view of what a Container solution offers. Instead we should take into account the entire ecosystem surround a solution including:

- Engine - features, security, resource management, resource requirements
- Orchestration
- Developer, and operations tool-sets
- Community, longevity and uptake
- Support

Because all container engines rely on the same kernel features they have some basic characteristics in common:

- a primary benefit is the isolation and encapsulation of application dependent libraries enabling different OS and library versions to be sandwiched together on the same underlying host
- the image sizes are dramatically reduced compared to other virtualisation technologies (to as little as a few MB or the size of a single statically linked executable)

⁶ <https://grsecurity.net>

⁷ Informative reference:
<https://www.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon>

- the start-up times are slashed (there is no OS boot time required - just the process(es) is started)
- and the operational density is increased considerably because of the removal of running a kernel and supporting OS processes per execution unit
- Applications can run on top of divergent hosts, and in heterogeneous environments
- Identical execution environments can be shared through exchange of images - applications take on some of the characteristics of data - they are shipped to the platform for processing
- they require root privileges to create images, and launch containers (excluding the most modern kernels - see runC later) irrespective of what privileges the container actually runs with. This is because of the kernel features and interactions required such as creating namespaces, cgroups, mounting (not including fuse) file-systems require root privileges.

It needs to be noted that even though containers provide encapsulation, they do not shelter the process from kernel dependencies so if for instance your machine learning algorithm uses the GPU then the libraries inside the container must be compatible with the hosts underlying abstraction for the GPU device.

In the SDP Pipeline use case, it is not necessary for the end users to have direct access to the container engine as there is the concept of the execution framework used to develop the execution graph and perform the scheduling of the pipeline components. This can be used to control the security profile of containers and will mitigate a lot of the security concerns around containerisation which would clear the way for the use of Docker (or any other kernel based Containerisation Engine) as opposed to a fully virtualised environment.

9.5. Container Engine Evaluation

Choosing the container engine has wide ranging implications for the SDP as it has implications on the selection for all the adjoining Middleware services, in particular the container orchestration layer.

It also has an effect on the software architecture for the pipeline processes, and the SDLC.

The following table is a comparison of features important to the SDP, focusing on health of the related project, feature completeness, and fit for SDP purpose. The three Container Engines are:

- [Intel Clear Containers](https://clearlinux.org/containers) ⁸ - a drop in KVM/QEMU based replacement for the default runC driver for Docker, providing full virtualisation (now part of Kata Containers)
- Singularity - a command line based executor for launching squashfs file-system based images, born from an inhouse project at Berkeley Labs, now run by [SyLabs](https://www.sylabs.io/) ⁹
- Docker - the leading open source container engine solution based on the [Moby Project](https://mobyproject.org/) ¹⁰

⁸ <https://clearlinux.org/containers>

⁹ <https://www.sylabs.io/>

¹⁰ <https://mobyproject.org/>

These have been chosen as leading solutions in their own niche - full virtualisation (Clear Containers), science oriented (Singularity), and full featured market leader (Docker).

Criteria	Clear Containers	Singularity	Docker
Standards compliant	Unclear what standards are intended to be supported but currently supports the Open Container Initiative (OCI). Appears to aim to support Docker images.	Unclear what standards are intended to be supported. Aims to support Linux OS images, and the import of Docker Images	Yes (but is commercially oriented) Good open development process with a community charter and governance
Application Container Support	Yes	Yes	Yes
Container portability (across engines)	Yes - only as far as Intel virtualisation technology is portable.	Yes - only through file-system export Restricted to Linux	Yes Supports Linux, Mac, and Windows
License	Apache 2.0	modified BSD 3	Apache 2.0
Project maturity, stability	Low	Low	Mid
Project health	Low - small but active community	Mid - is growing in HPC field	High
Modularity and extensibility	Low - does not have a defined API for 3rd party integrator	Low - does not have a defined API for 3rd party integrator	High - Docker is highly componentized as evidenced by the breaking up of the code base into separately supported open source projects. There is a plugin architecture available for networking, storage, authentication and scheduling.
Interoperability	High - the aim of the project is to provide an alternative runtime environment that is standards compliant - this so far has targeted Docker (since 1.12+) and Kubernetes	High in terms of managing target workloads, and environments Low in terms of integration potential with 3rd party applications	High - because of high configurability can run almost any workloads, and has a well supported API across many programming languages and frameworks

Community adoption and support	<p>Low - cautious adoption - no known use within research or academia.</p> <p>Responsive community support from core developers.</p> <p>No commercial support</p>	<p>Low - cautious adoption in HPC and research</p> <p>Limited support through fledgling community (community is helpful and responsive)</p> <p>No commercial support</p>	<p>High - wide spread use in open source, and gaining a lot of traction in the commercial sector. Not much recorded use in HPC, although has been supported by HPC vendors for 3-4 years eg: IBM</p> <p>Great focus and attention to detail on the user experience</p> <p>Good support through community channels, commercial support available, and there is a partner programme for extended support</p>
Documentation	<p>Mid - good architectural overview, and installation instructions. Limited information on detailed tuning and operation.</p>	<p>Low - documentation is hard to find and not well connected up, some good examples provided but needs to expand on the more subtle use cases</p>	<p>High - good quality searchable documentation with many examples both inside and out of the community</p>
Feature complete	<p>Low</p>	<p>Low</p>	<p>Mid - Orchestration layer has not caught up with Engine features</p>
Installation	<p>Easy, well packaged for supported environments and OSes, with well managed dependencies</p>	<p>Easy compilation and compact with few dependencies</p>	<p>Easy, well packaged for supported environments and OSes, with well managed dependencies</p>
Cost of ownership	<p>Mid - complex product, requiring detailed knowledge to administer.</p> <p>No license fees</p>	<p>Low - simplicity of features and ease of installation help make administration overheads low.</p> <p>No license fees</p>	<p>Mid - complex product, requiring detailed knowledge to administer.</p> <p>No license fees</p>
Security/isolation	<p>High - isolation based on KVM/QEMU</p>	<p>High - built in single mode based on requesting user - can only remove permissions from initial set.</p> <p>The singularity process invokes suid to get escalation in order to set the user environment before process exec</p> <p>root access required to build images</p> <p>There are some concerns about the use of chroot() and the mount</p>	<p>High (using AppArmor, SELinux, GRSEC, SECCOMP in conjunction with user namespaces, and capabilities) - requires the user to apply permissions so can control external threat but cannot control user behaviour</p> <p>Docker daemon is run as root in order to have sufficient rights to set the user environment</p> <p>API access can be controlled by plugins and SSL client certificates</p>

		namespaces (comments from runC rootless developer, needs further investigation)	
Resource management	None	None - leaves it to the batch system that invokes it	High - through cgroups
Networking	No host networking available	Relies on host network stack	Options - host, bridge, overlay, macvlan/ipvlan, 3rd party
File-system for boot image	As for Docker	Loopback devices and directory structures	CoW union file-systems - AUFS, BTRFS, Overlay(2), DeviceMapper
Development tools support	As for Docker	Low - command line interface only	High - command line, and libraries available for all major languages, and frameworks
SDLC tools integration eg: CI/CD, Project Management, and Source Code Control	As for Docker	Low	Mid
Direct access to host resources required for HPC	High	High	High
Orchestration Support	As for Docker and Kubernetes	Slurm, OpenMPI built in - can work with any scheduler that can execute a command line process	Docker Swarm, Kubernetes, Mesos, Nomad - can work with any scheduler that can execute a command line process or make a REST API call (Dask, DALiuGE, etc)
Logging, audit, monitoring	As for Docker	None	High - Docker includes drivers for various log management integration as well as good telemetry support in the daemon and via 3rd party apps eg: cAdvisor.
Desktop apps	Low	High - in default mode user has automatic access to all devices and current home directory, and /tmp	High - convoluted process to ensure that correct objects are mounted into container
MPI support	As for Docker	High - good integration and support from Open MPI	Low - MPI application can be run but there is no direct support and no recipes for shared ORTE daemon issue

GPU and other device support	vGPUs mapped into containers (GPU no. limited to 8)	High	High
Performance and Scalability	<p>Per container VM process overheads, and kernel overheads brokering IO access.</p> <p>All in addition to Docker overheads.</p>	<p>Single image so fractionally faster than Docker</p> <p>Loopback devices used for mount - 225 limit, and must be careful not to call write syncs as there are penalties</p> <p>Fixed sized EXT3 file-system images - must be careful to write to mounted spaces</p> <p>Mixture of shell scripts and C executables will incur startup costs</p>	<p>Union file-system is slightly slower</p> <p>Docker daemon is always running, producing approximately 0.2 vCPU(core) overhead</p> <p>Written in GO which has produced a lightweight and fast application architecture</p>
Configurability	High	Mid - configurable within it's defined niche	High - highly configurable to cover as wide as possible set of use cases
Privilege escalation. These are both using the same mechanisms with similar attack profiles i.e. any privileged code can have bugs/flaws that can be exploited.	High degree of kernel isolation.	Singularity escalates permissions to perform some setup operations, then drops permissions again before executing the container app	Docker runs as a privileged daemon that then sets up and runs containers with the appropriate security profile. Docker enables both privileged and unprivileged container execution, so it is up to the user to ensure that the appropriate invocations are made
Repositories	As for Docker	<p>Portable unit is a file, so any file distribution mechanism can be used.</p> <p>New community hub with the idea not to just publish containers but to publish reproducible research. In the near future will be reliant on Docker community hubs</p>	<p>Docker community hub, and other major hubs available eg: gcr, quay.io</p> <p>Private repository for internal hosting</p>

Images	As for Docker	<p>Single file, the image build process is aimed at creating a complete OS each time. These images get large, so as an alternative Singularity can read from a directory - eg: distribution via CVMFS(caching).</p> <p>Other formats: .tar, .tar.gz, .tar.bz2, cpio, cpio.gz</p>	<p>Images are layered, so they are built up and shared between divergent child images. build process will create a full OS image or single statically linked binary in a single layer. Efficient images that parallel pull operations. Simple web server caching in front of registries can be used for parallel cluster deployment</p> <p>Images can be exported as tar files complete with layers and manifest, or as a file-system tar file (unioned layers directory export)</p>
Foreground and Daemon processing	foreground and background can be explicitly controlled including attaching tty, and handling stdin/out/err	Default mode is foreground, but if executed process daemonizes, then singularity will drop into the background	foreground and background can be explicitly controlled including attaching tty, and handling stdin/out/err

Table 1 - Clear Containers vs Singularity vs Docker

Evaluating the criteria and assessing the comparisons, the following conclusions can be drawn:

- Clear Containers provide the highest level of isolation and security but at a significant performance penalty
- Singularity has specific support for MPI and Slurm but lacks the depth of community and product support crucially in the Container Orchestration and Cloud Native space
- Docker has the greatest fit for purpose and as a project has the greatest stability out of all the offerings

Both Docker Swarm and Kubernetes use Docker as the default Container Engine.

It is also important to consider the value of open source and the strength of the community surrounding a project. All projects are open source, but the size of the Docker project ([32K commits and 1,680 contributors](#)) and it's contributing developer team is enormous in comparison to Singularity ([2K commits 34 contributors](#)). Other signs of maturity are that Docker has a community charter, and governance with contributions and representation in industry standardisation, whereas Singularity has little or none. Singularity is also still at the stage where it is essentially in the hands one individual (the [creator](#)), so has no known succession or continuity plan. Docker has a commercial partnership plan forging relationships and garnering support from the [business sector](#) as well as in the [standards initiatives](#) .

10. Value of a Container driven pipeline in HPC

In a typical HPC environment, the applications run on a 'bare metal' host without abstraction layers between the application and the specialised hardware resources. The optimal execution environment for the processing pipeline requires the following:

- Fine grained scheduling within a compute node - eg: CPU/device pinning, and quotas
- Kernel parameter tuning per pipeline component
- Direct access to the host network interface
- Direct access to shared host memory and high speed storage devices

10.1. Selecting the point of Integration

Selecting the integration point between the Execution Control (the management layer responsible for orchestrating Execution Frameworks) and the platform is key. This defines the distribution of management complexity between the Execution Control and the Commodity software of the platform. The more functionality that can be shifted from Execution Control into the underlying platform, the less that needs to be bespoke software developed and maintained by the SDP project.

The following integration points are available for Execution Control:

- host OS - bespoke application running on bare metal. All scheduling and resource management controlled by Execution Control
- container engine - bespoke application encapsulated in containers running on bare metal. All scheduling and resource management controlled by Execution Control
- container orchestration - bespoke application encapsulated in containers running on bare metal. Scheduling and resource management split between the Execution Control and the orchestration layer
- scheduler - Execution Control behaves as a scheduling plugin that defines dependencies and resource requirements and then hands over full control to the scheduling agent of the container orchestration layer to determine resource placement and job allocation

10.2. To Orchestrate or not to Orchestrate?

Given that Execution Control defines a requirement for a custom scheduling and control solution, it could be argued that there is no need for an Orchestration solution. However, Execution Control still has a need to deploy and manage the custom scheduling and control solution (think [DALiUGe](https://dfms.readthedocs.io/en/latest/)¹¹ master/agents) as well as other software components and services such as:

- Pipeline processing components
- Monitoring agents
- Log aggregators
- Debug and trace tools

¹¹ <https://dfms.readthedocs.io/en/latest/>

These software components need a deployment and upgrade strategy. This could be handled by the Platform Management layer (Node OS deployment and management eg: Ansible) but this would be a less flexible solution than enabling a container based solution where any of these elements can be broadcast onto the platform as an isolated application component rather than a systems administration task.

Additionally, the custom scheduling and control solution needs a deployment strategy and the solution architecture can take advantage of the container deployment and failure management features that an orchestration solution provides to fulfil this.

10.3. Common Requirements

There are a number of capabilities that require consideration when evaluating containerisation and container orchestration services for the SDP. These requirements fall outside of comparing one solution to another, because they are more focused on what adoption of the Containerisation paradigm means.

10.3.1. Integrated Software Delivery Life Cycle

Within Containerisation, the opportunities for operational efficiency are not just confined to the supporting compute resources. There are arguably more significant benefits throughout the entire Software Delivery Life-cycle:

- A container will run identically on the developer's desktop, the testing environments, and the production environment because it carries all its own dependencies in terms of code, and libraries. Development is not necessarily confined to a specialised platform requirement (except dedicated/specialised hardware resources).
- Because containers are self contained, the software dependencies for the container are isolated from the host environment meaning that the management, and patching of the platform is entirely independent from the container software life-cycle
- Designers and developers of containerised applications can operate with much less demand and coordination placed on the infrastructure support teams as the point of integration is the high level abstraction of the container interface – not the individual application binding to the OS environment
- There is far greater opportunity to innovate through reduced platform dependency restrictions ie. there are less library, or software version restrictions placed on container designers and developers due to operational infrastructure management concerns and the natural tendency for atrophy of long running projects
- As container images are immutable, testability and reproducibility are a baked in concept

Even with these advantages, there are some restrictions that will be required for the SDP Compute Pipeline. As the performance requirements will be incredibly tight especially in the areas of real-time processing and the initial data ingestion from the CSP, it will be important to develop standards for pipeline software architecture and implementation.

These standards should cover important concepts such as:

- Storage handling

- Network interfacing
- Signal handling
- Logging
- instrumentation
- Container minimisation
- Container/Capability configuration
- Common Container Registry with a Library of optimised common base images
- Deployment registry with automatic container testing, profiling, security analysis, and version management
- Developer guides

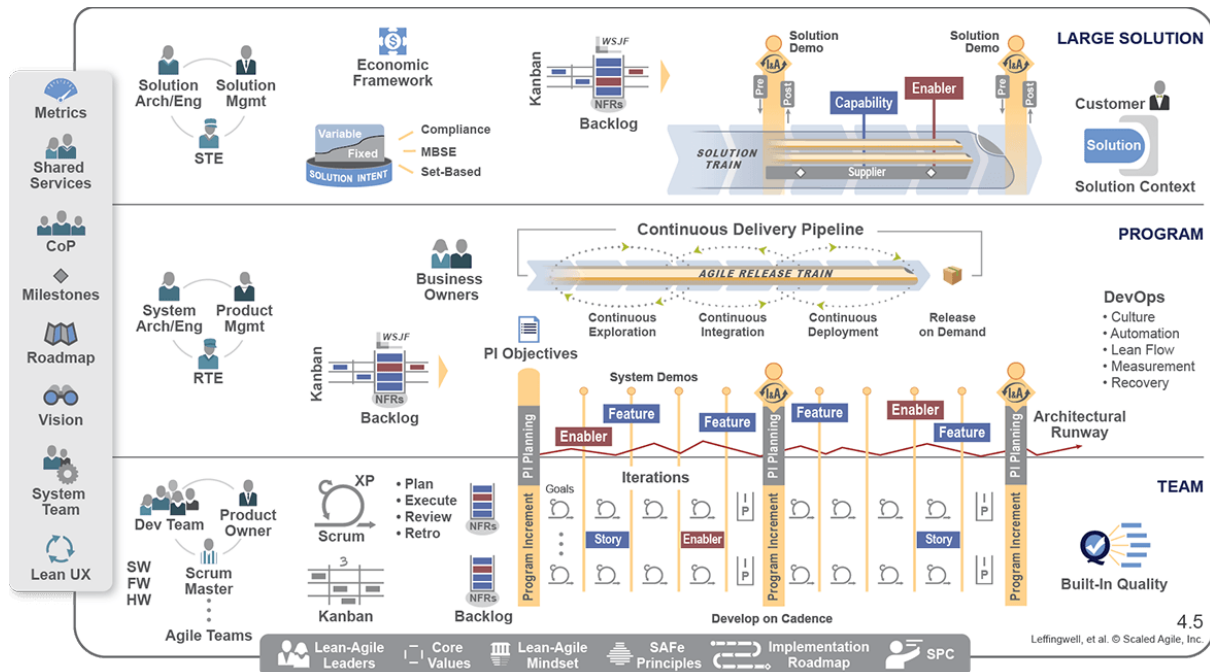


Figure 3 - The Software Development Life Cycle ¹²

In order to support the propagation of a high standard of software development optimised for the SDP operating environment, it is important that the necessary processes and tool chain are implemented and form the gateway to quality control and user acceptance for the SDP Pipelines.

Capabilities required are:

- Agile software project management
- Source code repository
- CI - automated testing and quality control
- CD - automated deployment
- High performance container registry
- Automated standards checking eg: Python PEP008
- Container Profiling
- Container security analysis

¹² Source: <https://www.scaledagileframework.com>

11. Orchestration Options

As part of assessing the industry offerings in the cloud native orchestration services space Docker Swarm, DCOS (Mesosphere), and Kubernetes have been reviewed.

11.1. Evaluation Matrix

The criteria for evaluation of cloud native orchestration services are as follows:

Criteria	DCOS	Docker Swarm	Kubernetes
COTS status is the solution Common as in widely used and widely understood, and is the solution 'Off The Shelf' in the sense that it requires little or no customising to achieve the requirements for the SDP	Is COTS, but will require heavy customisation to support complex scheduling, specialised devices and storage	Is COTS, but will require heavy customisation to support complex scheduling, specialised devices and storage	Is COTS but will require customisation to support specialised devices and storage
open source is the solution open source - this can fall into many categories and license types. It is necessary that the licensing is aligned with a core community license such as the GPLv3, AGPL, BSD, or Apache licenses. These licenses ensure the widest possible unrestricted use in the context of the SDP requirements. It is also important to note the licensing strategy of the solution provider are there are often uses of 'open core' type licensing vs fully open source. Open core often hides key features and functions away in commercial extensions to the core solution rendering it partially crippled or unfit for purpose unless the commercial extensions are purchased	open source - open core. Enterprise edition extensions offered for AAAI, networking and multi-tenancy, hybrid cloud and support	open source - open core. Enterprise edition extensions offered for AAAI, registry caching, networking, service deployment	open source
Licensing the licensing model - including open source - must fit the requirements, and cost objectives. This is especially important with dealing with 'per server or per seat' conditions or licensing tied to transactional volume or storage where	Predominantly Apache 2.0 - but some contributions covered by other open licenses	Apache 2.0 - but some contributions covered by other open licenses	Apache 2.0

the licensing cost penalties can grow quickly as the SDP capacity increases			
<p>Community health</p> <p>Is there a community behind the solution, and what is the state of that community.</p> <p>Does it have longevity and vigour - is there evidence of healthy responsiveness to dealing with issues</p> <p>Is there a roadmap, and are the feature directions in-line with SDP projected requirements</p> <p>Are there a variety of larger organisations involved with a long track record of open source involvement not unduly influenced by commercial objects</p> <p>What is the track record of use within the various scientific communities especially those that are data and compute intensive such as bio-informatics, Physics, Astronomy etc</p>	<p>Active for 7 years - Mesos - 14.4K commits, 269 contributors - Apache foundation project</p> <p>DC/OS - 3.6K commits, 111 contributors</p> <p>Marathon - 6.2K commits, 254 contributors</p> <p>Major users include DELL/EMC, HP, Cisco. Available on Azure, GCP, AWS</p> <p>Scientific users include: Meerkat SDP uses Mesos</p>	<p>Active for 5 years - 41.8K commits, 1.8K contributors</p> <p>Major users include: Expedia, Gsk, Splunk. Difficult to know full user base as unique user metrics are not available, but the "Hello, world" container example image "busybox" has had in excess of 1Billion pulls in the last 3 years</p> <p>Scientific users include: None known at this time.</p>	<p>Active for 4 years - 68.5K commits, 1.7K contributors</p> <p>Major users include Alibaba, Canonical, Samsung, SAP, OpenShift, Google, rackspace.</p> <p>Scientific users include: General fusion, and IHME¹³, CERN, OpenAI</p>
<p>Stability/maturity/longevity</p> <p>Are regular stable releases produced</p> <p>Is there a track record (3-5 years is good) of solid releases and well handled release procedures including upgrade management and continuity between releases</p>	<p>Mesos (core DC/OS) comes out of a project at UC Berkeley, and dates back prior to 2010.</p> <p>Major point releases are approximately every 6 months, with minor point releases (bug fixes/security) approximately monthly.</p>	<p>Stable releases every 3 months, with security and bug fix releases in between</p> <p>Core container support has been in the Linux kernel since 2006</p>	<p>Minor releases every 4 months since July 2015, with frequent point releases (monthly) for bug fixes and security. Supported by Google, and derived from internal "Borg" project with 15 years of development</p>
<p>Standards compliance</p> <p>If the solution operates in a well defined field where there are standards then it is compliant including:</p> <ul style="list-style-type: none"> statutory requirements 	<p>Supports the OCI container runtime and image format standards</p>	<p>Supports the OCI container runtime and image format standards</p>	<p>Supports the OCI container runtime and image format standards, and Cloud Native Computing Foundation standards for service offering</p>

¹³ <http://navops.io/ihme-case-study.html>

<ul style="list-style-type: none"> communication standards and protocols algorithmic standards software packaging, and delivery 			
<p>Feature complete</p> <p>Feature coverage within a given problem space includes covering the sufficient features that could be reasonably expected in that domain avoiding unnecessary additional solution implementations</p>	<p>Is complete for the average use case</p>	<p>Is complete for the average use case</p>	<p>Is complete for the extended use cases</p>
<p>Interoperability/Integration with 3rd party software</p> <p>What are the options and how well does the solution integrate with other software and solutions - are there existing integration packages, and libraries and is there good support for the standard protocols used in the problem space. Do the integration support the performance criteria of the SDP.</p> <p>What is the viability of the community around these components.</p>	<p>REST based API for automation integration</p> <p>Provides ability to integrate 3rd party logging and monitoring solutions, but does not turn off the duplication (no delegation).</p>	<p>REST based API for automation integration</p> <p>Extensive integration for logging, monitoring, storage, and network providers</p>	<p>Plugin and REST based API architecture from the ground up which is supporting a community driven enhancement ecosystem</p> <p>Extensive integration for logging, monitoring, storage, and network providers</p>
<p>Language and framework support</p> <p>Does the solution support the programming languages, libraries and frameworks necessary for the SDP in the solutions implementation context.</p> <p>What is the support of the community around these components.</p>	<p>Application deployment supports any language or framework that will fit in a container</p> <p>Mesos framework plugins are typically developed in Java</p>	<p>Application deployment supports any language or framework that will fit in a container</p> <p>Plugin development in Go</p>	<p>Application deployment supports any language or framework that will fit in a container</p> <p>Plugin development mainly in Go and Python, but can be in other languages supporting the REST based API</p>
<p>Documentation/Support</p> <p>Are the solution and supporting component integrations documented to a high standard including support for both users and developers. Does the documentation effort have a good history of evolving well with the solution development.</p>	<p>Yes - to the expected level of an ASF project</p>	<p>Very high quality documentation and training</p>	<p>Very high quality documentation and training</p>

<p>Performance/efficiency</p> <p>Is the solution designed and built with performance and efficiency in mind - ensuring that scaling and resource utilisation is optimised</p> <p>component model with independently scaling units</p> <p>reuse of components across different deployment schemes such as partitioned and multi-tenanted</p> <p>deployment of only the used functionality - reduced software footprint</p>	<p>Highly scalable - agent processes rely on Docker in common configuration and also incorporate JVM (overheads). Duplicates monitoring and logging processes with own scheme</p>	<p>Moderately scalable, and relies on the thin dockerd agent</p>	<p>Highly scalable - agent processes rely on Docker in common configuration. All agents in Go (low overheads)</p>
<p>Fit for purpose</p> <p>Does the solution comfortably meet all the use cases for the SDP, without undue compromise or work around</p>	<p>Functionality wise, DCOS is fit for purpose, but the framework overhead could be prohibitive and requires plugin development (resource and management and scheduling)</p>	<p>Is the lowest overhead solution with the least resource management and scheduling capabilities</p>	<p>Greatest fit for purpose due to moderate overheads (agent components over Docker) and feature coverage in resource management and scheduling as well as monitoring and logging integration</p>
<p>Fault tolerance/resilience/recovery</p> <p>What are the fault tolerance, resilience, and recovery qualities of the solution. How do they match the requirements of the SDP, and what are the impacts on integration with other solutions and design components</p>	<p>DCOS has resilience and fault tolerance built in using ZooKeeper, and a multi-master topology</p>	<p>Multi-master elections, and cluster recovery capabilities</p>	<p>Multi-master elections, and cluster recovery capabilities</p>
<p>Modularity/Extensibility</p> <p>Is it possible to extend the solution easily/effectively/cheaply to meet SDP requirements</p>	<p>It is likely that the scheduling capabilities do not extend adequately to cover the dynamic allocation of processing block affinity/anti-affinity requirements. This would necessitate the development of a custom scheduler plugin which is a</p>	<p>It is likely that the scheduling capabilities do not extend adequately to cover the dynamic allocation of processing block affinity/anti-affinity requirements. This would necessitate the development of a custom scheduler plugin which is a</p>	<p>Customisation is likely to centre around device and storage specific requirements for the SDP. Extended resource management and scheduling functionality should be adequate</p>

	complex and expensive task	complex and expensive task	
<p>Configurability</p> <p>Is the solution flexibly configurable to meet different implementation patterns required by the SDP. Configurability should reduce the burden of customisation of the solution to a level of acceptable cost dependent on the value that the solution brings to the SDP.</p>	Moderate	Moderate	High
<p>Security</p> <p>Does the solution meet the operational security requirements of the SDP:</p> <ul style="list-style-type: none"> authorised access restrictions intrusion protection and detection secrets and configuration handling best practice safety and isolation - protects itself and/or connected components from systemic harm from failure or intrusion 	<p>Auth and ACL - Yes</p> <p>Intrusion protection and detection - no</p> <p>secrets management - no</p> <p>safety and isolation - yes</p>	<p>Auth and ACL - Yes</p> <p>Intrusion protection and detection - no</p> <p>secrets management - yes</p> <p>safety and isolation - yes</p>	<p>Auth and ACL - Yes</p> <p>Intrusion protection and detection - no</p> <p>secrets management - yes</p> <p>safety and isolation - yes</p>
<p>Partitioned Management</p> <p>Can the service be partitioned, monitored, and managed in such a way as to support a multi-tenanted environment</p>	Enterprise edition	No	Yes
<p>Scalability</p> <p>efficient scaling of service to projected SDP utilisation levels</p>	Yes	Unknown	Yes
<p>Management/diagnostics tools</p> <p>Does the solution provide local and remote administration tools. What is the quality of diagnostic and troubleshooting tooling.</p>	<p>task diagnostic tools are adequate to the level of identifying and locating task failures with first level diagnostics.</p> <p>Subsequent detailed troubleshooting of errors/failures is a custom exercise</p>	Rudimentary. Enterprise edition unknown	Well defined diagnostic process, with detailed audit trail

<p>Logging/monitoring/accounting</p> <p>What kind of facilities does the solution have for logging, monitoring, and accounting.</p> <p>Does it integrate with standard protocols and tools and how does it integrate with the SDP logging/monitoring/accounting architecture - look for little or no integration costs</p> <p>What are the resource overheads like - will the necessary features introduce significant costs</p>	<p>All tasks can have a health check routine added with periodic checking rules.</p> <p>DCOS metrics, logs, and task specific logs can be plugged into ElasticStack, but this requires duplication of effort as DC/OS must maintain the agent communication flow.</p> <p>The basic DCOS monitoring and logging is insufficient for SDP needs</p>	<p>All tasks can have a health check routine added with periodic checking rules.</p> <p>Metrics, logs, and service specific logs can be plugged into ElasticStack</p>	<p>All tasks can have a health check routine added with periodic checking rules.</p> <p>Metrics, logs, and object specific logs can be plugged into ElasticStack</p>
<p>Cost - implementation/maintenance/running</p> <p>Understand the solution life-cycle costs including implementation, maintenance and running.</p> <p>How often does it need upgrading/patching how much time does it take, is there service disruption</p> <p>What are the resource overheads</p>	<p>Upgraded every 6 months</p> <p>Rolling updates can be performed on participating nodes. Each node/master must be taken out of service.</p> <p>DCOS agent software consumes approximately 2 cores and 1GB RAM</p>	<p>Upgraded every 4 months</p> <p>Rolling updates can be performed on participating nodes. Each node/master must be taken out of service.</p> <p>Docker daemon software consumes approximately 1 cores and 1GB RAM</p>	<p>Upgraded every 3 months</p> <p>Rolling updates can be performed on participating nodes. Each node/master must be taken out of service.</p> <p>Agent software consumes approximately 1 cores and 1GB RAM</p>
<p>Future prospects (eg: is this a foundational technology in support of future innovations)</p> <p>Is this core or niche technology. Are there aggregate benefits from dependant use from other solutions</p> <p>Is the solution aligned with strategic technology directions, and planned future innovation</p>	<p>DCOS seeks to address a use case for managing resource and task allocation at an aggregated infrastructure level - this is a growth area for hyperscale and HPC.</p> <p>However, its market is under threat from Kubernetes which has much higher adoption and flexibility</p>	<p>The aim for Docker has shifted from trying to claim the Container Orchestration market to being primarily a Container platform service provider including supporting Kubernetes.</p>	<p>Kubernetes has reached the point of being the dominant Container Orchestration engine, and is still rapidly building out its feature roadmap. It is the foundation of service providers including Docker, AWS, AKS, EKS, and OpenShift. Kubernetes appears to be positioned to form the basis next</p>

			generation composite cloud services
<p>Priority</p> <p>Priority defined by importance within the SDP architecture - used to prioritise effort expended on investigation and developing recommendations</p>	<p>High - scheduling and resource management is a key and complex feature of the processing handler interface to platform inventory management, and has serious implications for hardware utilisation and running costs</p>	<p>High - scheduling and resource management is a key and complex feature of the processing handler interface to platform inventory management, and has serious implications for hardware utilisation and running costs</p>	<p>High - scheduling and resource management is a key and complex feature of the processing handler interface to platform inventory management, and has serious implications for hardware utilisation and running costs</p>
<p>Integrated Execution Framework, Storage Orchestration, Software Defined Infrastructure management</p>	<p>Can integrate with existing execution frameworks (as demonstrated with Dask).</p> <p>Local host directories can be mounted, and external persistent volume management is performed using Rexray (supports many drivers eg: Ceph, Cinder etc.)</p> <p>Multiple network attachments can be added to tasks</p> <p>No inherent SDI management</p>	<p>Can integrate with existing execution frameworks (as demonstrated with Dask)</p> <p>Volume and network drivers.</p> <p>Docker compose framework for defining complex services.</p>	<p>Can integrate with existing execution frameworks (as demonstrated with Dask)</p> <p>Storage classes, and Persistent Volumes.</p> <p>Network policies based on namespace and Pod definitions.</p> <p>Complex resource descriptor language and Helm templating system.</p>
<p>Integration with Scheduler</p>	<p>Comes with options for out of the box schedulers, and ability to plugin user defined</p>	<p>Single Scheduler</p>	<p>Comes with options for out of the box schedulers, and ability to plugin user defined</p>
<p>Maintain multiple partitioned environments</p>	<p>Can be achieved through resource labelling, and roles but is not easily dynamic</p>	<p>No</p>	<p>Namespaced environments, and PodPresets for resource channelling</p>

Maintain multiple partitioned job allocations within an environment	Yes	Yes	Yes
Heterogeneous Infrastructure aware	Existing scheduler is aware of: Networks, CPU, GPU, Storage, Mem allocations.	Simple node label allocation	Scheduler can track and allocate mem, CPU, specialised devices, and storage
Service Discovery	High - DNS injection for services	High - core engine features automatic linking to services with DNS, and host file injection	High - DNS add on listens to service registration in etcd. Host file and ENV var injection.
Storage Orchestration	Relies on node local mounting, and bind mounting to containers	Volume drivers	Storage classes, persistent volumes, multi-mount.
Task Deployment and Release management	Moderate -rolling restarts for upgrades	High - rollout and rollback support, with failure modes and rolling update policies	High - rollout and rollback support, with rolling update policies - canary deployments
Self-healing	Restart/reschedule capabilities	High - Health checking capabilities, container restart policies	High - Health checking capabilities, container restart policies
Monitoring and logging	Internal logging and monitoring solution	High - logging drivers enable options for log aggregation. Log aggregation can also be managed through Logspout. Metrics interface built into Container Engine.	High - depends on underlying Docker capabilities. Also provides metrics endpoints for orchestrator, and logging goes to syslog/systemd - ElasticStack integration
Cluster Deployment tools	Moderate - dcos_generate_config .sh scripted installation process for deployment	High - Docker Swarm has a clear and stable toolset for cluster	High - kubeadm automates the cluster deployment. Has dependency on 3rd

		deployment that is easily automated (testing was done with Ansible)	party Pod network implementation (Calico good example)
Resource requirements and overheads	High - ZooKeeper, Docker, mesos agents.	Mid - Docker Swarm (latest Swarm Mode) runs as a single daemon and manages its own state machine using Raft between the masters. Requires additional monitoring and logging daemons per node (cadvisor, metricbeat)	Mid - Kubernetes runs multiple daemons per nodes (kubelet, proxy, pod network bridge). Separate master cluster. Master cluster state machine requires an etcd cluster. DNS requires additional distributed pod service. Requires additional monitoring and logging daemons per node (cadvisor, metricbeat)
Scalability	High - proven at 1000s of nodes and 100,000s of containers	High - proven at 1000s of nodes and 10,000s of containers	High - proven at 1000s of nodes and 100,000s of containers
Configuration Management	Low - environment variables, and config file injections	Mid - integration with own encrypted secret service. 3rd party integration with Vault by Hashicorp possible	Mid - integration with own and secret service that does not encrypt data. Possible 3rd party integration with Vault by Hashicorp. ConfigMap file injections

Table 4 - Container Orchestration evaluation matrix

11.2. Evaluation

Implementing any comprehensive platform management solution like a Container Orchestration engine is a major undertaking as it reaches into most aspects of the platform architecture and consequently the SDLC of components that the platform delivers. On the balance of the evaluation criteria, Kubernetes offers the greatest feature set for the least overheads and with the best outlook for future proofing.

Kubernetes has been evaluated twice - at version 1.6 when the Container Orchestration industry was in a closely fought contest between Docker Swarm, DCOS and Kubernetes, and 12 months later at version 1.10. In the intervening 12 months Kubernetes has emerged as the leader and has successfully redefined some of the fundamentals of software delivery and data centre management with the Cloud Native paradigm.

Developing software and services for the SDP on the Kubernetes platform will enable the SKA project to leverage the widest possible support for containerised software development, while in return developing software components to the broadest standards currently available - lowering the barrier to entry for reuse, integration, and uptake with integration partners ranging from core project developers, RSCs, and wider scientific communities - not just radio astronomy.

12. Kubernetes

The sheer weight of attention and contributors has seen the Kubernetes ecosystem grow at a staggering rate since Google first open sourced it in 2014. The core platform has stabilised considerably in 12 months with the difference between version 1.6, and 1.10 being marked with a maturity in release quality and API stability. Development focus now appears to be moving away from the primitives (Service, Pod, Deployment etc.) to higher order packaging with Helm Charts and Operators. Examples of this are rook.io¹⁴ which provides file, block, and object storage and KubeDB¹⁵ which provides a variety of database engines as a service both of which are implemented as first class Kubernetes objects.

12.1. Kubernetes Architecture

To understand the capabilities of Kubernetes for resource management and scheduling, it requires an understanding of architecture of the platform and the features that enable the platform to be tuned and customised to the SDPs requirements.

Kubernetes is typically thought of as fitting into the mould of a Container Orchestration services, however it has matured beyond this into a framework for delivering a Platform as a Service tuned to the local requirements. It manages a set of physical infrastructure resources in the form of Compute, Network and Storage, and intermediates these on behalf of the user for scheduling containerised workloads. The scheduling possibilities are complex ranging from cron jobs, and batches, to continuous services that are stateful or stateless, replicated, or daemonised.

The architecture is a master/slave arrangement. The master is in fact a multi-master solution (should be odd number ie. 1,3,5) where a raft consensus algorithm is used to manage consistency along with leader election.

¹⁴ <https://rook.io/>

¹⁵ <https://kubedb.com/>

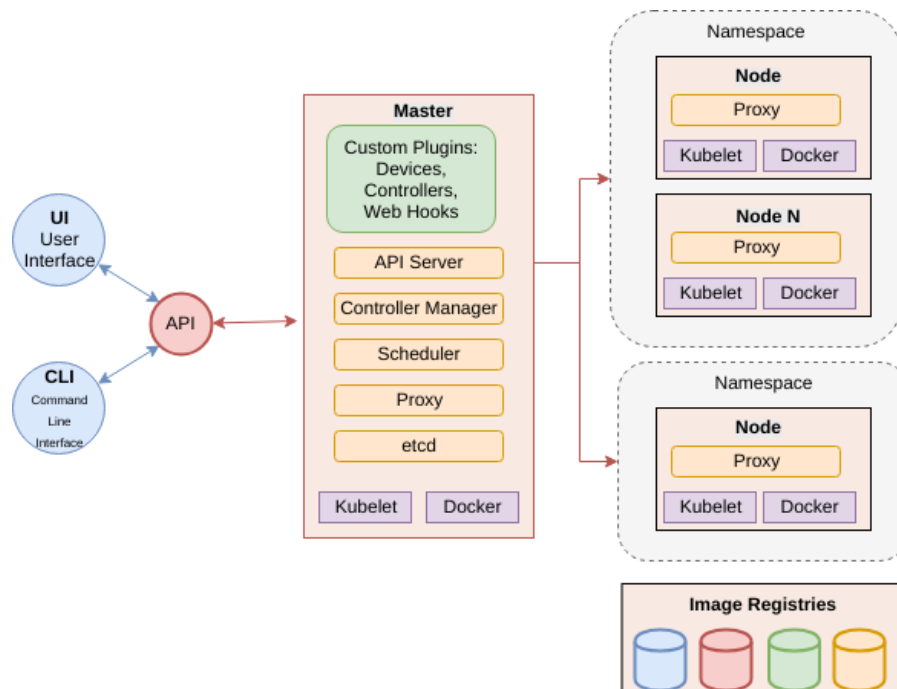


Figure 4 - The high level Kubernetes architecture

Kubernetes is made up of a suite of integrated components:

1. (client) user interfaces for monitoring and logging
2. (client) Command Line tools for administration and management (kubectl)
3. (master) an API Server which is the gateway of all client and command communication
4. (master) Controller manager directs the issuing of commands throughout the cluster to reach the desired state expressed through the API
5. (master) Scheduler calculates priority and placement of workloads with an awareness of cluster wide resource availability
6. (master) etcd replicated cluster state database
7. (all nodes) Proxy manages services and network policy enforcement on individual nodes
8. (all nodes) Kubelet is the primary node level agent throughout the Kubernetes cluster and is responsible for bootstrapping the entire framework, as well as executing Pod specifications by collating the necessary resources, and communicating with the underlying container engine. In the common use case, Kubernetes is installed on top of Docker which is used as the container engine and the interface with container registries. Registries are where the container images (packaged software) for running container instances are sourced from.
9. Namespace enable resources and quotas to be carved up so that workloads can be partitioned at the cluster, rack, node, and sub-node levels. Access and quota rights enforce policy with regard to deployment of this partitioned access at more granular levels of actions (create, read, update, delete), resource allocation (eg. memory, CPU, etc.) and Kubernetes Primitives (Pods, Services, storage classes etc.).

All components communicate over granular public APIs that are designed to provide a consistent view and behaviour giving maximum opportunity to program and extend the platform.

Kubernetes by virtue of the container engine integrates with all public and private container registries that subscribe to the Open Container Image specification (<https://www.opencontainers.org/>).

Overall, Kubernetes with containerisation provides a mechanism to decouple infrastructure deployment and management from resource allocation and workload scheduling. The unit of allocation is no longer a disk, adaptor, or server, it is now a unit of CPU, memory, network bandwidth, and storage capacity - each of a specific class.

12.2. Kubernetes Primitives

The Primitives are the fundamental building blocks that Kubernetes offers to build higher order applications and services. These fall into the following categories:

- Infrastructure
- Scheduling models
- Container packagers

An additional cross-cutting category is namespaces. Namespaces scope nearly all Kubernetes primitives enabling high level separation of workloads and resource allocations, as well as defining security contexts.

It should be remembered that all Kubernetes objects are described using a document descriptor (YAML/JSON) that is POSTed to the relevant API endpoint. These documents describe a desired state for the Object, and the Kubernetes platform will create, or migrate existing objects to match this. This desired state declarative approach underpins the entire operational management approach of the Kubernetes design.

Primitive	Description
Storage Classes and Persistent Volumes	<p>Storage classes represent different categories of storage which can be based on type, cost and location. Examples of this might be NFS, Parallel FS, Block Storage, raw device, temporary space, and ram device where the class maps to a specific driver for the kind of storage addressed.</p> <p>Storage classes can be used to define coarse grained data locality and desired performance characteristics.</p> <p>Persistent Volumes are allocations of storage within quota against a given Storage class. These allocations can persist across separate container executions, so can form the basis for long lived services such as databases, or in the case of the SDP for example, an observations visibility data.</p>
Services	<p>Services expose network ports from Pods in a variety of ways. These can be as a NodePort which is the same port number on every node in the cluster, ClusterIP - a designated IP address within the cluster address space, or over an external load balancer. The load balancer integration is typically used for bridging the underlying bare metal network services, or a cloud providers public fixed IP services.</p>

Ingress	Ingress controllers enable the specific mapping of an applications service/port specification to a load balanced URI. In this way, a collection of services can be mashed together to provide a single service in the style of a microservices architecture.
Network Policy	Network policies define the rules for how Pods are allowed to communicate with each other and the outside world.
Scheduling Models	Scheduling modes control the run-time behaviour of applications.
Job & CronJob	Jobs are one or more Pod deployments that run once and are expected to exit on conclusion. A CronJob is a scheduled Job with all the general characteristics of UNIX cron with additional dependency management, restart, retry, deadline and concurrency characteristics. This model is suited to the batch scheduling workloads of the SDP.
ReplicaSet	Are a mechanism for controlling the scaling factors for are Pods (where a Pod has not been declared within a Deployment).
Deployment	A Deployment is an amalgamation of a ReplicaSet, and Pod descriptors. Deployments are typically for describing steady-state services such as a typical web application or micro-services suite component. The ReplicaSet component describes the scaling characteristics, while the Pod component describes the container set. Manual scaling tools and the Horizontal Pod Autoscaler can be applied to the Deployment as a single coordinated unit to match resource allocation to load. This is also true for the rolling-update tools that enable roll-forward, and roll-back of Pod updates including Canary patterns for partial upgrade+test scenarios.
StatefulSet	StatefulSets are essentially the same as Deployments except persistent identifiers are carried through any Pod scheduling changes. The effect of this is to preserve things such as network connections, and relationships with persistent storage. It also provides some guarantees about the ordering of deployment, updates, and terminations, features that ideal for services like an RDBMS.
DaemonSet	DaemonSets ensure that an associated Pod runs on all nodes as specified by the nodeSelector (defaults to all). This is ideal for Node common services such as monitoring and logging, and distributed configuration management.
Autoscaling	<p>There are two kinds of autoscaling currently available with Kubernetes:</p> <ol style="list-style-type: none"> 1. Pod autoscaling - where a HorizontalPodAutoscaler object can be created that will watch specified metrics on the associated deployment replicaset and automatically scale up/down based on these triggers. 2. Cluster autoscaling - where Kubernetes will scale the cluster size up/down by communicating with the cloud provisioning API of the infrastructure service provider.

	<p>As resource management automatically driven by cluster metrics may not be desirable for Telescope Manager, these features may not be used strictly in an automated fashion, however the API and triggers of the AutoScaler could be used for manual control of the cluster size as a way of moving between power modes/states and gracefully resizing the cluster (node draining, workload relocation, protection of loads that cannot be moved before completion or profiles such as Pod disruption budget etc.) based on anticipated workloads defined in the observation schedule.</p> <p>The Kubernetes metrics API, and the Node status API give a ready made integration point for the Telescope Manager and Execution Control to understand the cluster state, workloads, and resource availability.</p>
Container Packager (Pod)	<p>There is only one container packager, and this is the Pod Spec which is used by all of the above Scheduling Models. The Pod Spec defines what containers will run within a Pod, what their network and storage inputs and outputs look like, and what other resources such as Memory, CPU, and device specific components are to be allocated. A Pod has a securityContext that can be defined that effectively controls the Capabilities, namespace and cgroup settings for the running containers. A Pod also can have a health service specification (livenessProbe) so that Kubernetes can understand whether remedial actions such as restarts need to be carried out.</p> <p>Containers in a Pod have a shared namespace and security context. How this works is Kubernetes creates an empty header container and then squashes an additional container into the same namespace for each container specified in the Pod Spec. This gives the containers in the Pod the same network stack. It also means that individual containers can be restarted within the Pod, without pulling down the entire Pod container suite.</p>

Table 2 - Kubernetes Primitives

12.3. Deployment Patterns

There are many ways that Kubernetes can be controlled, both from an interactive (user command line) and automated (API client libraries) contexts.

Deployment Pattern	Description
Ad Hoc	Most of the core Kubernetes objects (Pod, Replication Controller, Service, etc.) allow ad hoc creation via the kubectl command line client
Resource Descriptors	<p>Resource Descriptors are YAML files that adhere to the API specifications ¹⁶ for the various supported Kubernetes objects. Each file can contain one or more object definitions making it the primary mechanism for creating static recipes for application deployments.</p> <p>Resource Descriptors are executed by using the kubectl create, apply, update, and delete verbs</p>
Kubernetes API Clients	The client APIs ¹⁷ enable the embedding of Kubernetes management functionality into 3rd party applications. All of the APIs adhere to the Swagger.IO JSON documentation standard ¹⁸ , which can be accessed through the interactive /swagger-ui/ testing interface (which must be enabled on the kube-apiserver). These APIs are proxied through object abstractions in the client libraries for ease of use.
PodPreset	<p>PodPresets are Pod templates that act at the point of admission control using label selectors to sieve out incoming Pod descriptors and to then apply defaults before containers are launched. Any Pods (including those in Deployments) that are modified are flagged in the annotations.</p> <p>This facility opens a way, with no extra coding or component deployment, for the application of uniform defaults for labelling, resources, and configuration to be applied to application groups.</p>
Helm	Helm Charts provide a means for deploying complex applications using a meta templating system. Each component of an application suite can be defined with variable substitution anchors, that are replaced at runtime with values declared either on the command line

¹⁶ <https://kubernetes.io/docs/concepts/overview/kubernetes-api/>

¹⁷ <https://github.com/kubernetes-client/python>

¹⁸ <https://kubernetes.io/docs/concepts/overview/kubernetes-api/>

	<p>or in a values.yaml declaration document. Charts can be recursively nested, and helper functions that calculate variables or even blocks of YAML for template declarations can be incorporated.</p> <p>Helm Charts have a sophisticated meta-templating language, making it possible to create container fleet descriptors that are customised at runtime with configuration values. As with resource descriptors (which is what Helm Charts render as output), Helm charts are declarative specifying a desired state that the Kubernetes scheduler will calculate changes for. For example - specify a change in the number of replicas for a Deployment, and the scheduler will reap or spawn sufficient Pods to meet the directive.</p>
--	--

Table 3 - Kubernetes deployment patterns

12.4. Extending Kubernetes

Kubernetes is a complex and modular system that is designed from the ground up to be extensible. The [following diagram](#)¹⁹ illustrates the 7 ways in which Kubernetes can be turned into a custom platform.

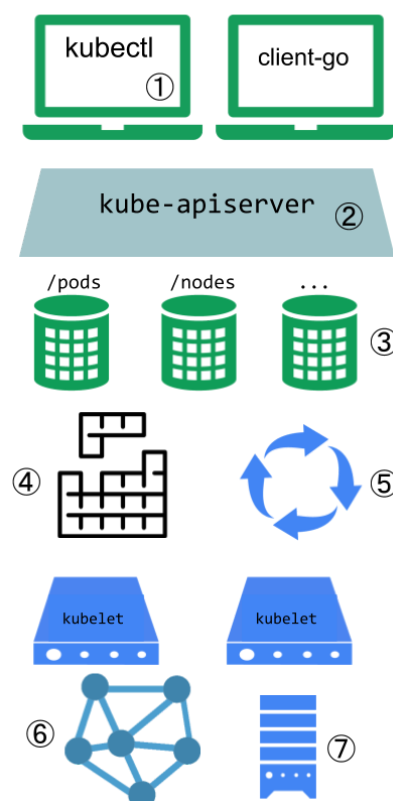


Figure 5 - Extension points for Kubernetes

¹⁹ Source: <https://kubernetes.io/docs/concepts/extend-kubernetes/extend-cluster/>

1. kubectl is the command line "front door" for interacting with the Kubernetes cluster. This can be extended with plugins that modify API Server requests such as headers for authentication (default is to use client/server certificates). The OpenStack Keystone extension does this on the client side which integrates with the corresponding server side component (webhook). There are the official client-go libraries which are used both internally to Kubernetes and externally to create custom automations.
2. The apiserver handles all requests both internally and externally. Several types of extension points in the apiserver enable hooks for authenticating requests, or blocking them based on their content, editing content, and handling object deletions. These are collectively called Admission Controllers.
3. The apiserver can be extended with custom resources using Custom Resource Definitions. These create user defined corresponding endpoints to the API complete with parameter descriptions and validation rules. These form the user interface to (5) Custom Controllers which will do the heavy lifting.
4. The Kubernetes scheduler calculates Pod placements at runtime. The default scheduler can be replaced or extended with a custom one.
5. Within Kubernetes, the implementation of handling different Resource types are implemented with Controllers. Together with Custom Resource Definitions, and the GoLang API framework, completely new resource types and their behaviour can be implemented eg: a Postgres database Custom Resource that automatically deploys a persistent volume claim, stateful set, and service to expose a PostgreSQL database.
6. Custom Network plugins that handle the Pod Network through implementing the CNI. Eg: Calico, and Kuryr (OpenStack). This includes the handling of applying network policies.
7. Storage plugins that attach to storage classes via provisioner identifiers handle the mounting of storage for the kubelet (root Kubernetes node daemon) ready to be mounted into Pods at creation time. These can be stacked with existing core handlers eg: a new Network FileSystem driver can expose shares through the NFS handler, or a new storage architecture could mount space onto a node and the expose this through the core host bind mount. Storage provisioners are available for many major existing solutions such as GlusterFS, NFS, CephFS, CephRBD, OpenStack Cinder.

12.4.1. Dynamic Admission Control

Admission Control WebHooks enable custom access control, and reformatting of requests as they flow through the API Server. Using this feature it is possible to completely rewrite and augment incoming API requests.

There are three admission control plugin points:

- static authentication and access control - this is what the OpenStack Keystone integration uses
- mutating webhook - returns API calls such as PATCH commands that modify the original resource descriptor eg: add containers to Pods, labels, default configuration, secrets etc.
- validating web hook - can give an OK/NOK for incoming API calls

Both mutating and validating webhooks can be configured to filter for particular resource types (eg: Pods, Services, etc.), and event types (CREATE, UPDATE, etc). As with PodPresets, these

features enable the injecting of configuration, labels, containers, and other associated resources into resources as they flow in to be scheduled. This could include specifying local data volume mounts, sidecar containers²⁰ for logging/monitoring/debugging, or injecting sensitive secret data such as service connection information and certificates at run-time.

12.4.2. Custom Resource Definitions

At the heart of extending Kubernetes is the ability to create new Resource types through defining completely new objects as first class resources registered in the API Server. These resources are available through all of the client integration options.

12.4.3. Custom Resource Definitions & Controllers = Operators

Combining CRDs with Controllers creates what are typically referred to as an Operator. These are aggregations of existing primitives that are put together and managed as a single entity. An excellent example of this is the [Postgres Operator](#) which creates, monitors, scales, and backs up PostgreSQL database instances as first class resources in Kubernetes.

How they work is that the Operator controller component registers itself with the API Server to handle API requests for a specific object namespace as registered via the CRD. The controller is completely custom code enabling a request to be translated into the necessary component actions to manipulate Kubernetes primitives that define the custom resource. Following the simple example above for a Dask Cluster, on receipt of a create request the controller will:

- create the meta object Daskcluster that can be tracked with `kubectl get daskcluster` etc.
- create a Deployment containing the replica set of 1 and pod descriptor for launching the Dask Scheduler container
- create a Deployment containing the replica set and pod descriptor for launching the Dask Worker container
- create a Service exposing Bokeh and the Scheduler if necessary

This mechanism lends itself well to the encapsulation of repeatable, transportable, and high value system processes that can be encapsulated in a custom construct that manages the object life cycle. For a relatively new feature in Kubernetes (Beta in 1.10 - April, 2018), there is already considerable support for complex resources that would ordinarily be managed in OpenStack including:

- [Rook](#)²¹ - file, block, and object storage services
- [KubeDB](#)²² - Postgres, MySQL, Redis, Elasticsearch, Memcached, MongoDB
- Elasticsearch - monitoring and logging
- Prometheus - monitoring, logging, and alerting
- Kafka - Data Queue

²⁰ Container with a separate task pushed into the same namespace

²¹ <https://github.com/rook/rook>

²² <https://kubedb.com/>

12.4.4. Custom Devices

Custom device plugins enable new hardware oriented resources to be registered for availability (including life-cycle management) with the Kubernetes scheduler inventory. This is used for the NVIDIA GPU, where the plugin will count the number of GPU devices available on each host (the plugin checks and reports on device status) and then makes them available as resources that can be requested when a Pod is launched. The device plugin is also consulted at the time the dependent container launches and has the opportunity to pass device configuration in the form of environment variables. This will enable the Pod container to auto-discover what it needs to utilise the device.

This can be extended to any real (eg: represent specialist devices such as GPU and FPGA, clocks, NIC) or virtual limited resource that requires resource accounting during Pod scheduling.

Device plugins can also be developed for cluster wide device concepts that are not tied to any particular node - for example an NVMe based enclosure that offers up FPGA devices.

12.4.5. Storage Plugins

Kubernetes storage plugins are fundamentally built around the POSIX file-system with the expected mount point to be either a host-path directory, block device, or network file-system. Storage plugins are realised in a number of ways:

- in-tree - eg: NFS, hostPath etc
- out-of-tree, FlexVolume, Container Storage Interface

There is wide support for third party storage services including:

- Gluster
- NFS
- Ceph RBD
- CephFS
- OpenStack Cinder (overlay for Ceph, LVM2 etc)
- Flocker

12.4.6. Network

Kubernetes requires a network plugin that acts as the interface to the Pod network. The Pod network is essentially the network plumbing that enables (or disables) the communication between Pods within a host, and across the cluster. The network plugin takes the Pod network commands issued by the Kubelet from the scheduler, via the Container Network Interface and converts these into native networking commands depending on what the underlying network provider is eg: OpenVSwitch or a proprietary router etc. The network plugins also enforce network policies which describe cluster rules about what Pods are allowed to communicate with cluster wide.

For Kubernetes running on OpenStack there are two leading options:

- flat layer 2 networking for all cluster nodes - Calico, which does not need to use an overlay
- OpenStack Neutron plugin Kuryr

Kubernetes networking also provides hooks for integration with external LoadBalancers, which are automatically mapped on creation of a type: LoadBalancer Service descriptor. Integration with OpenStack Octavia (haproxy based load balancing) is directly supported.

12.5. Kubernetes in HPC

HPC workloads vary widely with many types modeled on high parallelism and task times that can be very short (seconds) and long (minutes/hours/days). With the basic unit of work on Kubernetes being a Pod, the start up and tear down for extremely short lived tasks may not be appropriate for HPC workloads. However, given that the nature of Execution Frameworks such as MPI²³/SLURM²⁴ and DALiuGE²⁵ work on a principle of running their own master/slave control network, it is entirely practical to use Kubernetes as the scheduler that deploys the Execution Framework cluster, and then hand over job control to the SDP Execution Control component. In this way a single coherent view of fine grained resource availability is retained, but the entire cluster can handle mixed workloads.

²³ <https://www.open-mpi.org>

²⁴ <https://slurm.schedmd.com/>

²⁵ <https://daliuge.readthedocs.io/en/latest/>

13. Resource Management

This section consists of translating the capabilities of the Kubernetes platform into the resource management and scheduling necessary to support the SDP environment - pipeline processing, real-time, support services.

Resource	Description
Memory	Partial (virtual) memory allocation. Memory is allocated in bytes and requests can be expressed in Ki, Mi, Gi, Ti etc. The i suffix is for base two and can be omitted. If a container exceeds its memory allocation then depending on the scheduler, its Pod will be evicted.
CPU	Partial CPU allocation - virtual. 1 CPU is equal to 1 core hyper thread. These can be allocated down to the 0.001 partial unit or 1 millicpu. A container may be restarted for exceeding its allocation.
Ephemeral Storage	Temporary work space for containers are held in the uppermost and writeable layer in the union file-system. This space is typically mapped into /var/log and /var/lib/kubelet. The limits for consumption of this space can be specified by declaring the ephemeral-storage resource. When a container exceeds the specified limit then the Pod is evicted.
Devices - Extended Resources	Any Node or Cluster level resource that requires accounting but no further special handling (such as device plugins above), can be configured. Requests for these will be counted by the scheduler and used for determining Pod placement.

14. Service Discovery

Kubernetes maintains a cluster-wide DNS service that is automatically synchronised with Pod and Service resources so that resources can advertise their locality. A Service created with the name `daskscheduler` listening on Port 80 (name `http`) will be given two entries:

1. an A record pointing to the service name scoped by the namespace (default) - `daskscheduler.default.svc.cluster.local`
2. an SRV record similarly scoped for service port - `_http._tcp.daskscheduler.default.svc.cluster.local`

Pods are also plumbed into DNS with A record entries based on the hostname: and subdomain: Pod Spec configuration entries eg (assume hostname: `scheduler`, subdomain: `dask`): `scheduler.dask.default.svc.cluster.local`.

Pods consuming services advertised from other Pods can discover the services in two ways:

1. using DNS lookup based on prior understanding of names eg: Pod B knows that a service will be advertised at `postgres.default.svc.cluster.local` from Pod A
2. Environment variables pushed into the current container context, mapping all known services with IP, Port, and Protocol

This means inter-service connectivity can be discovered at container (re)start time.

Service discovery is a key mechanism for enabling applications to discover their environment, and supporting services at run time reducing the burden of explicitly configuring all aspects of each instance of a launching application. This is considered a core solution architecture component of container orchestration systems where service dependencies between containers are resolved by name or “handle” instead of requiring hard coding of addresses.

15. Scheduling

Scheduling in Kubernetes enables the resources of the entire cluster to be allocated using a fine grained model. These resources can be partitioned according to user policies, namespaces, and quotas. The default scheduler is a comprehensive rules processing engine that should be able to satisfy most needs, and then most subsequent custom requirements can be satisfied by using plugins (Admission Control, PodPresets) to tweak and reshape requests so that combined with the default scheduler the desired effect is created.

The primary mechanism for routing incoming tasks to execution is by having a labelling system throughout the cluster that reflects the segregation of workloads and types of resources required, coupled with Node and Pod affinity/anti-affinity rules. These are applied like a sieve to the available resources that the Scheduler keeps track of to determine if resources are available and where the Pod can be placed.

15.1. The Scheduling Process

The following diagram gives an example of how scheduling and resource management can work in Kubernetes, and how this can be applied to workloads for the SDP.

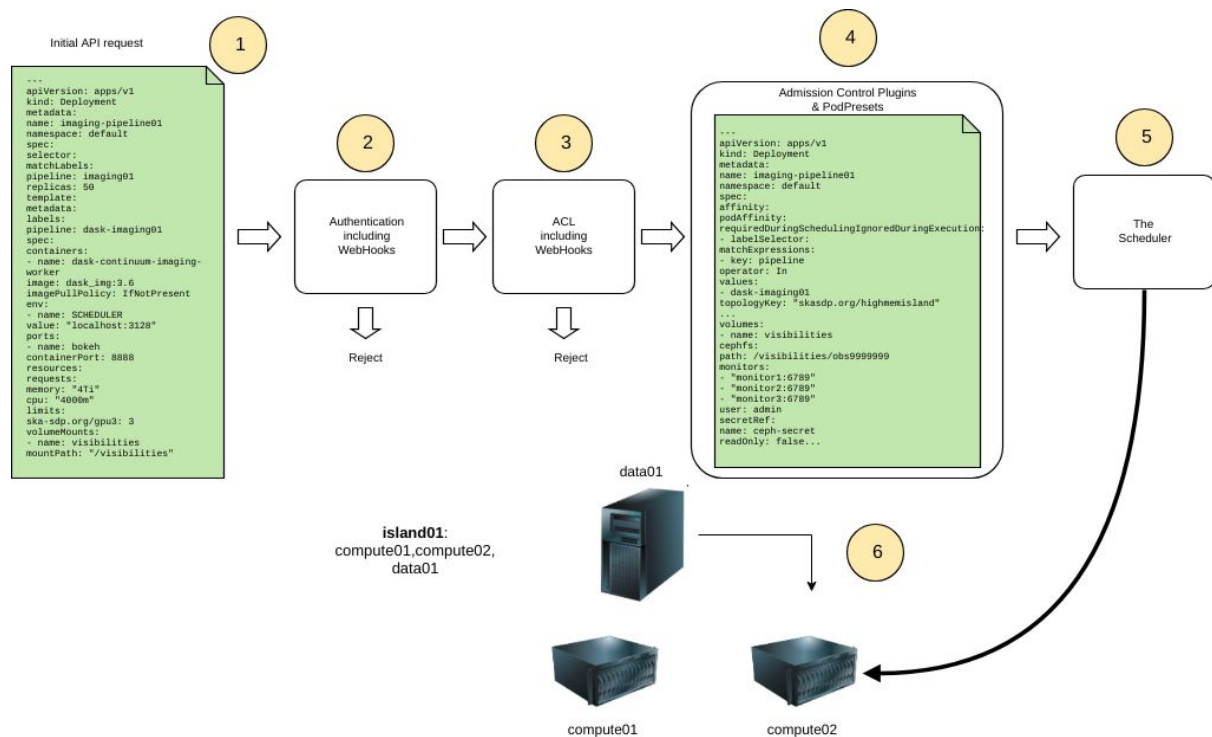


Figure 6 - Kubernetes scheduling process flow

1

A deployment descriptor is posted to the Kubernetes API Server describing how to run an Execution Framework component - in this case a [Dask Worker](#)²⁶. The worker container asks for 4 CPUs (2 cores x 2 hyper threads) and 4 Terabytes of memory, along with 3 SDP specific GPU devices. The container also specifies a volume mount for visibilities, but as yet there is no associated volumes definition. The deployment also asks for 50 replicas as this is a scale-out workload. The Pod has a high level description label pipeline: imaging01 that identifies what kind of task this is, and is used later to expand node placement rules, and to resolve data locality.

2

The incoming request passes through Authentication where webhook integration with OpenStack Keystone verifies the authentication token passed in the Authorization Header. An invalid token would see this request rejected (403) at this stage.

3

Access Control is used to verify that the user behind the request has the authority to create and/or update the incoming Deployment. The OpenStack webhook integration matches verb (actions) rule lists against the Keystone roles associated with the authenticated user.

4

Once past Authentication and ACL, the request is handed over to the Admission Control plugin. This is where a custom plugin can translate high level indicators like label pipeline: imaging01 into a physical location by looking up the SDP configuration database to see which data island the incoming visibilities are scheduled on, and where the nearest compute island resources are aligned with this to satisfy the request. Each Kubernetes minion (worker node) within a physical rack is tagged with labels that define data and compute island groupings. This enables data locality to be translated to persistent volume claims and resource availability to be translated into label and affinity definitions that the Scheduler will understand.

Note: The previous ingest pipeline will have landed the visibilities in something like CephFS. This volume/volumeMount arrangement could easily be swapped for object storage configuration passed as either configMaps or environment variables.

5

When the request reaches the Scheduler, it now has all the necessary information for the Scheduler to calculate whether the live resources are available, and where the Pod should be launched. Memory, CPU, ephemeral storage, and device availability is checked and then the Pod launch requests are passed off to the individual node kubelet daemons for actioning.

The affinity rules cluster all the Pod replicas in the same set of racks that represent the target compute island. The affinity and resource limit rules can be soft (preferredDuringSchedulingIgnoredDuringExecution) or hard (requiredDuringSchedulingIgnoredDuringExecution) enabling a choice to be made whether Pods can spill over to other racks in the event of lack of resource.

²⁶ <https://distributed.readthedocs.io/en/latest/worker.html>



Scheduler instructions are passed to the Kubelet daemon on each node. The Kubelet will ensure that the specified volumes are mounted and shared into the correct container mount points. Mounts can be shared between Pods so Pods can gain efficiencies through running and communicating on the same Node. The Kubelet sets up the necessary Pod networking rules. The Kubelet also hands over variables from device plugins, applies resource constraints, and then boots containers into the Pod.

16. Kubernetes: Cloud Native Implementation Patterns for the SDP

While the core function of the SDP is to process visibilities into images, there are a wide variety of workloads platform wide:

- platform shared services - databases (of all kinds), queues, reporting tools, Tango, SDP configuration, software configuration, software and container registries, software repositories, etc.
- standard cluster node profile services - monitoring and logging agents, debugging tools, OS housekeeping services, bootstrap processes such as Kubelet, device plugins, network plugin
- central cluster services - master server processes such as cluster state directory, DNS, controller, scheduler
- ad hoc workloads - development, testing, diagnostics, Continuous Integration/Continuous Deployment
- real time SDP processing - ingest, calibration
- batch SDP processing - image processing

These workloads will be dynamic as the cluster scales up and down to meet processing schedules and SDP conservation targets (resources are moved in and out of power conservation modes).

Kubernetes is well positioned to provide a dynamic programmable platform that can accommodate the different and partitioned workload requirements for the SDP. Kubernetes provides an air gap between the infrastructure services that require specialised and highly complex management practices and processes from the architecture, and DevOps requirements of design and implementation for SDP workloads. This enables common tasks such as allocation of storage, network and compute to be delegated to the SDP construction teams in support of their agile development processes, without losing control through quota and partitioning capabilities.

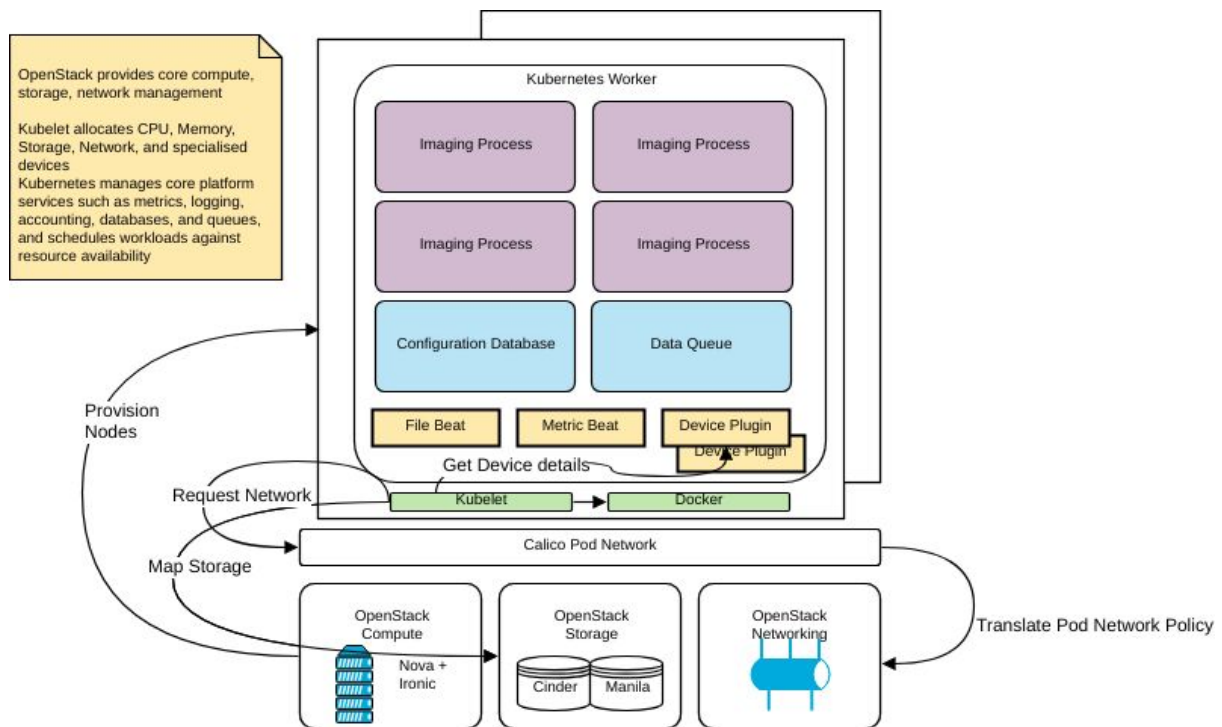


Figure 7 - *Kubernetes integrated with infrastructure*

16.1. Accessibility

As Kubernetes has evolved rapidly over the last 3 years, tasks that would normally be handled by OpenStack such as the provisioning and management of platform services like databases, search engines, logging, monitoring, accounting, and data queues are now available as first class resources. The associated communities have seen the benefits of developing the tools to support the full life-cycle within a containerised environment and that this technology is more rapidly developed, has a lower barrier to entry, and a wider audience than the counterparts developed as purely OpenStack sub-projects. The prime example of this is OpenStack Trove - the database as a service project that has been rapidly overtaken by the KubeDB project. Not only can these kinds of components be deployed on bare-metal Kubernetes, but also fit seamlessly into OpenStack Magnum (Kubernetes), Google Container Platform, OpenShift, Azure AKS, and AWS EKS.

From a developer perspective, these tool-sets can also be run on the desktop using the MiniKube environment²⁷. This universal access means that the platform is available to all to contribute to on practically any developer hardware platform which lowers the barrier to contribution within the SKA agile delivery teams, and externally from the wider research community. This extends even to specialised storage and device requirements where the interface with Kubernetes is abstracted away - an example of this is storage classes which are an

²⁷ <https://kubernetes.io/docs/setup/minikube/>

abstraction from the underlying storage driver - an OpenStack Cinder block device could be substituted by the hostPath implementation for development purposes.

16.2. Flexibility Satisfies Complexity

At the core of Kubernetes is Linux Containerisation as provided by the container run-time. This means that most features that are available for manipulating namespaces, cgroups, and capabilities are available to the Pod container specification. The run-times can be swapped on different nodes to provide different kernel level capabilities eg: Docker, libcontainerd, Rkt, runv, and clear containers, which provide different implementations of kernel based containerisation, and virtual server based containerisation. Being able to provide a mixed landscape enables workloads with different isolation, security, and kernel capability requirements to run in a single clustered environment, with these specialised low level requirements abstracted away from development.

At Pod run time resources can be shared between containers in Pods, and between Pods running on the same node, including:

- networks
- devices
- mountable volumes
- memory and IPC context can be shared enabling direct communication between containerised processes

Configuration is injected at run-time in the form of environment variables, so containers can discover what they are, and what services to communicate with at process boot time. This dynamic configuration can be extended with ConfigMap and Secret objects stored in the Kubernetes cluster directory, or with 3rd party operations configuration solutions like HashiCorp Consul and/or Vault to completely decouple management of configuration from code. This can be further supported with PodPresets objects that act as filters at the cluster Controller level to inject standard templates such as labels, default configuration pointers, sidecar containers, logging and monitoring directives - limited basically by imagination.

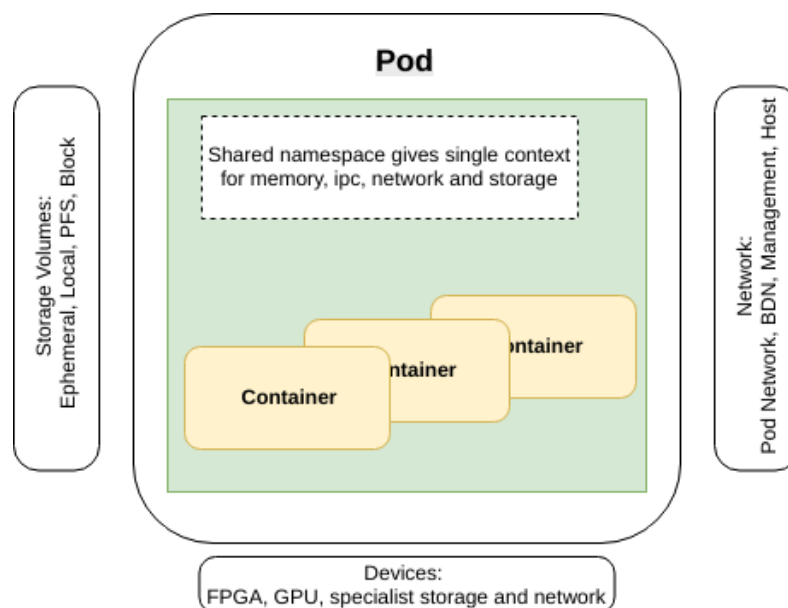


Figure 8 - *Anatomy of a Pod*

Kubernetes has the ability to integrate with all types of storage via storage classes:

- Ephemeral: temp local disk and memory
- Local: direct attached storage devices
- parallel file-systems: CephFS, Gluster, Lustre, BeeGFS
- Block: OpenStack Cinder - CephRBD, LVM2, and block device over NVMeoF - eg: NVMeSH (Excelero)
- Object Storage - Ceph RADOS, OpenStack Swift

NVMeoF block devices is an interesting alternative to all the others as it off loads the CPU burden, and has low latency. This could present visibilities and intermediate results as a Data Island in a separate enclosure that could potentially solve the mass synchronisation point latency issues by shared or remounting storage to the gather node.

16.3. Execution Framework Agnostic

Kubernetes does not dictate what Execution Framework the SDP will be required to use. Due to the flexibility of the scheduling modes, resource reservations and Pod structures, any present day framework can be deployed. Examples include:

- the ARL library and the (above) example of deploying a Dask cluster
- [Kube-openmpi](https://github.com/everpeace/kube-openmpi) ²⁸ is a set of Helm Charts, and Dockerfile image definitions to help get going with migrating an existing MPI based application to run on a Kubernetes cluster
- [Apache Spark](https://github.com/GoogleCloudPlatform/spark-on-k8s-operator) ²⁹ - yet another Operator that makes Spark infrastructure components first class resources on Kubernetes

16.4. Kubernetes Containerisation and SDLC

The SKA as a whole is planning on adopting the SAFe Agile development methodology, and core to this is the approach to the Software Development Life Cycle. Kubernetes integrated with a social coding platform like GitLab is purpose built to support this in the highly distributed project team environment that the SDP will have.

The Minikube single instance Kubernetes cluster will comfortably deploy on a laptop, where it will be possible for the SKA to provide a simple bootstrap script with a set of published container base images to ensure development environment consistency. These images can be published on a service like the GitLab integrated container registry, and the setup scripts can be provided as Helm Charts.

²⁸ <https://github.com/everpeace/kube-openmpi>

²⁹ <https://github.com/GoogleCloudPlatform/spark-on-k8s-operator>

GitLab provides the complete Social Coding environment with: Git repository, peer review, task management, release management, integrated CI/CD, and integrated container image registry. This will enable an environment where code contributions can be distributed with the appropriate peer review, and automated testing gates. This is central to defining the project culture for coding standards, quality, and documentation as well as providing mechanisms to enable hands-free deployments across integrated development, test, and production environments.

Kubernetes supports this project structure with dynamically configurable namespaced environment, authorisation and access control and quota policies.

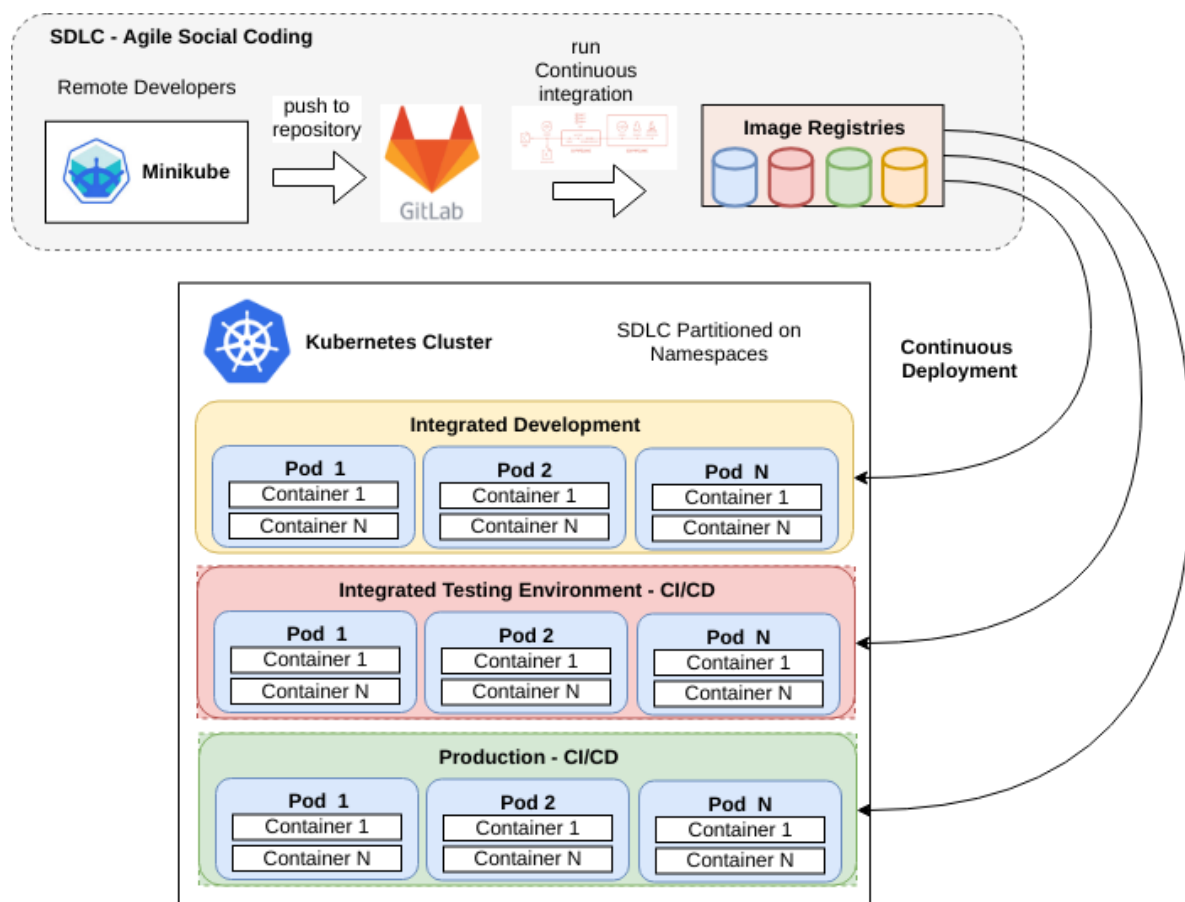


Figure 9 - Automated DevOps

GitLab CI/CD pipelines are fully integrated so reporting and visibility of outcomes live within the repository, fully linked to the commits, merge requests, issues, and manual interventions that triggered them

passed Pipeline #25683613 triggered 6 days ago by Piers Harding

fix deploy to work with remote

7 jobs from `master` in 39 minutes 59 seconds (queued for 1 second)

1065fcb9

Pipeline Jobs 7

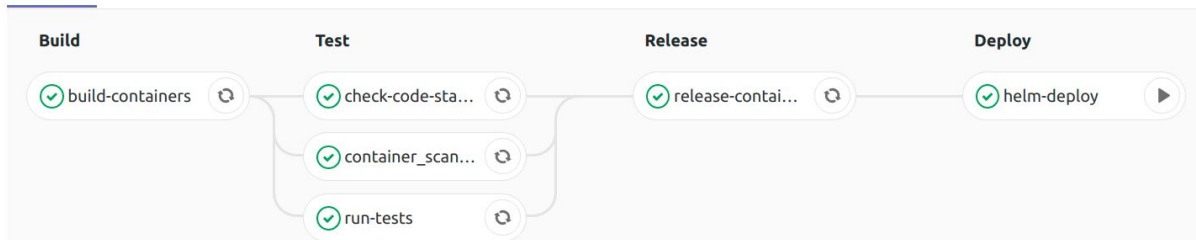


Figure 10 - GitLab flexible CI/CD pipelines

16.5. Applying the Patterns to the SDP

The SDP has the notion of an Execution Control which will take directives for managing the life cycle of pipelines in accordance with the observation scheduled, from the Telescope Manager. Execution Control is designed to be effectively Execution Framework agnostic, so will be able to set up, direct, and tear down jobs that are driven by Dask, DALiuGE, Apache Spark, Slurm, native MPI and any other candidate.

The most complex kinds of jobs for scheduling are the ones that require careful mapping of tasks to locality of resources. Taking the example of scheduling an imaging pipeline process, where a pipeline will benefit from:

- being located within nodes that have the fastest link to the storage of the related observations visibilities
- proximity to each other as intermediate results are shared through the scatter and gather processes aligned with the synchronisation points for major cycles
- access to model and calibration data updates

Storage proximity is important regardless of the implementation (PFS, local, Object, NVMeoF Block devices etc.) as communication paths are optimised in descending order of in node, within rack, and within rack clusters based on the network hops typically defined by physical switches.

The SDP has the concept of:

- Compute Island - a partitioned subset of compute nodes that can span one or more (partial) physical racks
- Data Island - a partitioned subset of storage that can span one more more (partial) storage appliances, and racks

The Network topology by convention will have top of rack switches that route traffic between nodes in a rack, and then a local area network that will map traffic between racks, and between racks and the wide area network. This is separated into three networks - lights out (infrastructure management), management and bulk data networks.

The following is one example of how these conditions and requirements can be mapped onto Kubernetes capabilities:

An Imaging pipeline will consist of many sub-tasks that will be coordinated by Execution Control. Each of these sub-tasks can be directly mapped to a Pod with a set of resource requirements. The job in total is linked to the ingested visibilities of the parent observation. This bulk data (a minimum of 14.5PB for 6 hours at 5.52Tb/s [RD01]) will either be landed and left on a hot buffer data island from the preceding real-time ingest process or pre-staged from the cold buffer to the hot buffer data island ready for the imaging pipeline. Associated with this data island based on the underlying storage locality is a lookup table of compute node and rack locality indicating increasing network cost (distance) from the data island. Using this lookup table and associated resource availability information in Kubernetes (each node advertises available resources through the common API), Execution control can describe and nominally reserve a virtual compute island for the pipeline.

Once the resources are reserved, Execution Control can now schedule the tasks against the compute island with the necessary memory, CPU, and device requirements. This can be handed over to Kubernetes where an Admission Control plugin can take the compute island label definition and map this to affinity and anti-affinity rules for placement in the cluster providing the basis for scheduling. The Admission Control will also inject the necessary default configuration for:

- monitoring and logging
- database and data queue connections
- persistent volume claims, and scratch space

At this point, the Controller then hands over to the Kubernetes scheduler which uses the rules and task structure to:

- place a primary task on a nominated head node in the compute island
- launch remaining tasks that have the affinity/anti-affinity rules that provide gravity to propel placement onto the compute island

The primary task provides a centre of gravity that will pull associated pipeline tasks onto the nominated set of nodes/racks through affinity rules. Affinity/Anti-affinity rules will ensure that task density is spread evenly across the nodes where tasks that need to share resources (memory, pipes, temporary storage etc.) are placed on the same node, and tasks that would compete for resources are placed apart. Device plugins are used to manage resource counts and configuration for GPUs/FPGAs by the scheduler.

16.6. Kubernetes Overheads

[Kubernetes at scale](#)³⁰ - ie. 1000 nodes (currently to a maximum of 5000) - will require an HA environment with a minimum of 3 master nodes. Within this, an HA etcd database needs to be deployed as the internal state database, and can reside on the master nodes. Currently hardware requirements for master nodes at the upper end are estimated based on 500+ nodes:

- Google Cloud machine types with the [n1-standard-32](#)³¹ - which is essentially 32 VCPU, 120 GB RAM
- AWS with the [c4.8xlarge](#)³² - which is 36VCPU, and 60GB RAM

Disk space requirements are minimal in the order of 2x400GB SSD RAID1.

Detailed profiling of the Kubernetes Master services on a 1000 node cluster have been carried out by Marek Grabowski for the [Kubernetes project](#)³³:

99 percentile: container memory (MB)	cpu (cores)	
"etcd-server-e2e-test-gmarek-master/etcd-container"	0.312	1792.36
"etcd-server-events-e2e-test-gmarek-master/etcd-container"	0.095	1846.36
"flannel-server-e2e-test-gmarek-master/etcd-container"	0.006	73.48
"flannel-server-e2e-test-gmarek-master/flannel-container"	0.001	134.51
"flannel-server-e2e-test-gmarek-master/flannel-server-helper"	0.000	1.61
"kube-apiserver-e2e-test-gmarek-master/kube-apiserver"	5.924	4285.18
"kube-controller-manager-e2e-test-gmarek-master/kube-controller-manager"	3.570	4507.16
"kube-scheduler-e2e-test-gmarek-master/kube-scheduler"	1.758	5541.92

³⁰ <https://kubernetes.io/docs/setup/cluster-large/>

³¹ <https://cloud.google.com/compute/docs/machine-types>

³² <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/compute-optimized-instances.html>

³³ <https://github.com/kubernetes/kubernetes/issues/21500>

17. In Summary

A Cloud Native architecture delivered on Kubernetes provides a holistic approach, with an immersive environment that abstracts design and development from infrastructure concerns to a large degree. This gives a single point of control for user interaction, scheduling and resource allocation, which in turn will give the SDP greater control over access, resource consumption, and capacity planning.

Benefits include:

- reduces platform awareness required by execution control. Execution control can schedule on higher level abstract concepts such as compute island, cluster capacity, node/resource type, and let the platform translate this into a run-time schedule
- platform schedules and maintains housekeeping activities such as monitoring and logging distributed as containers in daemonsets, and periodic tasks as CronJobs. This approach brings application container style SDLC control to common OS level services
- provides abstract model for managing core platform services and software building components such as RDBMS, Search, and Data Queues with wider community support through operators than is enjoyed by OpenStack (supersede OpenStack derivative projects with operators for things like Postgres, MySQL, Kafka, ElasticSearch etc.)
- abstraction from resource management - details of storage are handled by infrastructure and abstracted by classification, decoupling implementation from application design and development. This enables design, development, and deployment to scale from the desktop up to the live cluster
- idempotency of deployable unit with Containerisation
- encapsulation of dependencies with Containerisation
- uniform management of different task classes - all daemons, jobs, and services treated exactly the same
- speed of deployment with unified file-system and image layers
- isolation of workloads through partitioning features such as namespaces and affinity rules
- finer grained packing against resources (sub-node level)
- automated scaling and load balancing
- automated recovery against node failure - restart/rescheduling comes for free
- resource locality aware scheduling (data, device, storage, memory, CPU)
- fine grained resource accounting (data, device, storage, memory, CPU) by the scheduler
- ubiquity in the wider IT sector - Kubernetes is a multi-vendor long term platform supported by Google, AWS, Azure, OpenStack, RedHat etc.
- specialised storage can be modelled with operators - eg: NFS or BeeOnD
- create first class (CRD) objects that encapsulate common higher order objects eg: for storage with particular placement constraints
- create jobs/services with affinity for the storage object/placement
- create dependent/related jobs with affinity for prior jobs to get locality/proximity aware based placement
- integrates seamlessly with OpenStack - services requiring persistent storage can be backed by OpenStack storage primitives such as block devices
- Service Catalog - external provisioner integration

- draw on a wealth of packaged software solutions encapsulated in Helm Charts provided by community contribution. SDP can publish a catalogue of approved and customised applications deployments (services and pipelines) delivering self-bootstrapping coding and software standards to the wider project teams by example
- join clusters and burst resources using cluster federation

The SDP has a requirement to schedule a variety of workloads against complex and varied resources. These concerns are common to all workloads and in particular to Execution Control and the Execution Frameworks that will be supported for the image processing pipelines.

Resource management and scheduling, by the very nature of the problem require a single point of registration and accounting so that there is a clear picture of what resources are available to avoid over and under commitment due to incomplete or inaccurate information. Kubernetes provides this overarching fine grained control along with all the other described flexibility and integration potential making it currently the best of breed solution for delivering Platform Services for the SDP.