H2020-EINFRA-2017

EINFRA-21-2017 - Platform-driven e-infrastructure innovation

DARE [777413] "Delivering Agile Research Excellence on European e-Infrastructures"

Operational Requirements and Guidelines I

Project Reference No	777413 — DARE — H2020-EINFRA-2017 / EINFRA-21-2017
Deliverable	D5.3 Operational Requirements and Guidelines I
Work package	WP5: Platform Operation and Maintenance
Tasks involved	T5.1: Provision of Relevant Cloud Infrastructure T5.2: Provision of Pre-release Testbeds T5.3: Deployment Strategy and Platform Operation T5.4 Operational Requirements Specifications and Roadmap
Type	R: Document, report
Dissemination Level	PU = Public
Due Date	31/12/2018
Submission Date	31/12/2018
Status	Draft – v5
Editor(s)	Byron Georgantopoulos (GRNET), Stavros Sachtouris (GRNET)
Contributor(s)	
Reviewer(s)	Iraklis Klampanos (NCSRD)

Document description	This deliverable reports on the set of guidelines and best practices for managing and maintaining a DARE platform deployment on the cloud. The DARE components' development and deployment lifecycle have to follow certain processes in order for them to be successfully and smoothly integrated within the DARE platform. The second release of D5.3 after the third project year will elaborate on the final guidelines followed to ensure successful operation of the deployed DARE platform.

Document Revision History

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
1	31/10/2018	Initial Structure	B. Georgantopoulos (GRNET)
2	01/12/2018	Outline	Stavros Sachtouris (GRNET)
3	19/12/2018	Content	Stavros Sachtouris (GRNET)
4	23/12/2018	Introduction and conclusion	Stavros Sachtouris (GRNET)
5	31/12/2018	Finalisation	Iraklis Klampanos (NCSRD)

Executive Summary

This deliverable reports on the set of guidelines and best practices for managing and maintaining a DARE platform deployment on the cloud. Reviews of the most mature and widely adopted tools in the industry are combined with best practices and guidelines for deploying the DARE platform, with regards to the complexity and requirements in computation, storage, networking, security, isolation and user management. Furthermore, a container-based multi-tiered environment based on a cloud-deployed Kubernetes is examined.

Table of Contents

List of Terms and Abbreviations

Abbreviation	Definition
DARE	Delivering Agile Research Environments
K8s	Kubernetes
WaaS	Workflow as a Service
P4	Protected Pervasive Persistent Provenance
C4	Common Conceptual Core Catalogue
PE	Process Element
VM	Virtual Machine
AWS	Amazon Web Services
EOSC	European Open Science Cloud
EUDAT	European Association of Databases for Education and Training
CRI	Container Runtime Interface
OCI	Open Container Initiative
CNM	Container Network Model
CNI	Container Network Interface
REST or RESTful API	REpresentational State Transfer APplication Interface
DBMS	DataBase Management System
EFS	Elastic File System
HDFS	HaDoop File System
POSIX	Portable Operating System Interface
ELK	Elasticsearch Logstash Kibana
PPK	Private-Public Key
LDAP	Lightweight Directory Access Protocol
SAML	Single Assertion Markup Language
RBAC	Role-Based Access Control
TLS	Transport Layer Security
OIDC	Open ID Connect

1 Introduction

The aim of this document is to provide guidance, best practices and suggested procedures regarding the deployment, maintenance and support of the DARE platform [1]. At this point, the architecture of the platform has been shaped, but many important details are still under discussion. When dealing with established components or issues for which there is consensus, this document provides specific guidelines. When dealing with issues and parts of the architecture that are still undecided, lists of options, industry standards and best practices are provided.

Each chapter in this document is organized according to the layers of the platform: infrastructure, container platform, core services, process elements (PEs). In some cases the core services will be examined in conjunction with either the container platform or the PEs. Readers looking for one of this aspects can read only the parts of each chapter related to their interest. For instance, if someone is looking for guidelines on how to deploy Kubernetes for DARE, they should skip to the sub-chapters named "platform" or "container" and only take a quick look at "infrastructure".

1.1 The DARE platform

At the top level, the DARE platform comprises three services:

1. Workflows-as-a-Service (WaaS) that performs the actions required by a data-driven application domain.
2. Protected Pervasive Persistent Provenance (P4) that keeps records of all that is done and supports multiple uses of those records.
3. Common Conceptual Core Catalogue (C4) that helps researchers build and share an integrated view of multiple information sources, by holding all of the items of interest to them. It enables users, projects, groups and communities to build and re-use their information space and holds the information needed by software and specialists.

The DARE platform [1] must be supported by systems which can store and transfer large amounts of data, can maintain various access levels on many kinds of metadata and can deploy processes on diverse technologies. From an operational perspective, the platform interfaces include some user authorization and authentication, which are based on C4. Complex relations between users, operations and data are handled by C4 and P4 services. The details of their functionality are out of the scope of this document, which focuses on how these services are deployed and maintained instead.

It is important to distinguish PEs from the DARE services. PEs live in C4 and, from a systemic point of view, their instances are executed in an isolated environment. They are relevant to end users, groups or organizations, while services are controlled by the DARE system operators. Still, they are both software services and they can be treated in a similar manner e.g., if they run in containers.

Last but not least, information over the execution of PEs must be collected e.g., for provenance. Containers must be logged as they are executed, and the results must be stored and provided through a querying mechanism.

1.2 Containers

Modern cloud technologies offer computing power, networking and storage capabilities, while containers allow the execution of isolated processes which can be powerfully controlled and orchestrated. The combination of the two allow the construction of multilayer platforms with powerful features such as replicated services, isolation of processes, shared storage, security and safety tailored to the needs of each service and refined user roles. The key is to choose and configure a container orchestration or automation platform over a well provisioned cloud infrastructure.

Here is a list of the most mature and widely used such platforms:

- **Kubernetes**: [2] a portable, extensible open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It is the evolution of Google's container deployment system and supports many container runtimes. It provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, logging, and monitoring. However, Kubernetes is not monolithic, and these default solutions are optional and pluggable. Kubernetes provides the building blocks for building developer platforms, but preserves user choice and flexibility. At this moment Kubernetes is the strongest candidate for the DARE platform.
- **Docker swarm**: [3] a clustering and scheduling tool for Docker containers. With Swarm, IT administrators and developers can establish and manage a cluster of Docker nodes as a single virtual system. It is easy to set up, it greatly simplifies operations like scaling and replication, and requires minimal configuration (if any). This is achieved by making default choices on issues like underlying networking, process scheduling and level of isolation.
- **Apache Mesos with Chronos and Marathon**: [4], [5], [6] a cluster manager that provides efficient resource isolation and sharing across distributed applications or frameworks. Mesos is a open source software originally developed at the University of California at Berkeley. It sits between the application layer and the operating system and makes it easier to deploy and manage applications in large-scale clustered environments more efficiently. It is always accompanied by additional services which take care of scheduling or zookeeping. Mesos can be configured to deploy containers on a cluster, using Marathon for permanent services and Chronos for short lived tasks.

Kubernetes (or K8s) is the most feature-rich offering. With monitoring, user management and scheduling customization included [7], it is a all-in-one solution. What's more, it allows running DARE core services and PEs on the same platform, as containers. It provides fine-grained control regarding container connectivity. It also allows the definition of refined user roles and the implementation of established authorization and authentication methods. Finally, the information-reach API can be used by DARE services to collect PE execution data.

Kubernetes is more suitable than swarm, because it is open source, configurable and extensible. It is also more suitable than Mesos, because it is better supported with tools and documentation, and has a lot of much needed features included. Thus, on the following chapters we assume that DARE is built on K8s or a container platform with similar characteristics.

2 Provisioning

2.1 Infrastructure provisioning

The goal of provisioning the basic infrastructure is managing the size and state of actual or virtual compute and storage resources . The term “compute resources” refers to virtual machines and their CPU cores, RAM memory and disk volumes, virtual private networks and public IPs attached on those VMs, as well as storage volumes which may be shared, detached and reattached. These resources are typically allocated on an IaaS cloud provider (e.g., ~okeanos [8], AWS [9], EOSC[10], Azure) as well as on premises systems.

A good example of an appropriate cloud infrastructure is EOSC, an EUDAT initiative aiming to provide a collection of federated cloud services by integrating various existing and future solutions in research data infrastructures. It is a relatively new solution (the pilot face was completed recently) and therefore it hasn't reach the maturity levels of other vendors, it provides a wide range of inter-operable cloud applications and infrastructure interfaces, thus making it a good fit for DARE.

Another example is the ~okeanos IaaS, which can be managed through a web UI, a command line client, a python library (kamaki [12]) or an ansible [13] script. After logging in ~okeanos, they can request the creation of a number of VMs, by specifying the number of CPU cores, RAM, disk size and operating system. They can also reserve and attach public IPs (version 4 and/or 6) or create and attach detachable disk volumes and configure accessibility to the created VMs (login/password and/or Public-Private Key authentication).

The size of required resources is estimated in advance and adjusted dynamically as the system evolves, based on statistics regarding CPU utilization, network traffic, disk I/Os, etc. In the case of a cloud, this data is collected by the cloud vendors and is accessible through APIs, but it can also be collected by the platform installed on top of the resources (e.g., Kubernetes [2], Mesos [4], Docker swarm [3]) or by monitoring tools (e.g., Prometheus [14], Grafana [15]). This issue is further discussed in the “Monitoring and accounting” section.

2.1.1 Automation

An abundance of automation tools can be used for automatic infrastructure provisioning e.g., ansible [13], puppet [16], juju [17] and others. Automation tools are provided with a few structured configuration files (e.g., yaml files for ansible) which contain either instructions (e.g., “create a VM with features X”) or the desired state of the system (e.g., “a VM with features X is up and running”). Communication between automation tools and the cloud is established and, through the cloud API, a number of operations is performed until the system described in the configuration files is created and functional.

These tools primarily focus on automating the deployment and configuration of multilayered services and they can orchestrate complex systems on pre-existing infrastructure. Therefore, it is not mandatory to orchestrate the infrastructure layer with automating tools or with the same tools and scripts as the

ones used for the rest of the service. The question of including with or excluding the infrastructure from the automation scripts does not have a definitive answer, as both approaches have pros and cons.

The main advantage of a full-stack automation script is the ability to timely recover dysfunctional systems, especially in case of failures. If this is the case, the operators can just run the scripts and have a similar system available as soon as possible. Adjusting and evolving the infrastructure can also be handled in a quick and organized fashion, by just updating the desired state of the system. For instance, in order to add a monitoring server on a well automated service, one may just have to increase a numeric variable (assuming there is one concerning the number of monitoring servers to be deployed) in a configuration file and re-run the ansible or puppet script. What's more, migration from cluster to cluster can be much faster. Last but not least, infrastructure automation helps testing and distributing the platform e.g., to developers or partners maintaining their own infrastructure.

The main disadvantage is the possible cloud lock in. Automation scripts require special support for each cloud API, therefore migration to different clouds may require the adjustment of the infrastructure scripts. For instance, the initial system may be deployed on an academic OpenStack [18] cluster, but it is later required to also be deployed on a commercial cloud like AWS. Even if the same procedures work on both clouds, differences in billing and accounting as well as the technical approach on the cloud backend may renter a deployment procedure sub-optimal or cost ineffective. For instance, although it is possible to deploy with the same script on Synnefo/~okeanos [19] (which is OpenStack compatible) and on an OpenStack cloud, the two stacks differ in the way they manage user access and quotas.

Some operators may prefer to separate the infrastructure automation from the rest of the platform, in order to have control over their deployment. By keeping the two procedures separated, a system can appeal to operators who prefer to handle their infrastructure using their own tools. Although this doesn't count as a disadvantage of full automation, it does count as an advantage of keeping the two procedures independent to each other.

2.1.2 Capacity planning and scaling

Infrastructure planning can be preemptive or adjustable. The preemptive approach dictates meticulous estimations and the reservation of an abundance of resources early on, so that the system can handle the highest expected utilization. The adjustable (or agile) approach suggests a small initial system which is adjusted to the actual usage automatically or manually, as the system evolves and the user base increases. In practice, successful systems are planned using a combination of the aforementioned approaches. They are built with well estimated but conservative initial requirements and are equipped with tools and processes to easily adjust if their actual needs increase or decrease.

It is strongly advised to estimate the amount of resources i.e., computing (CPU, RAM, disk volumes), network (IPs, private networks, traffic) and storage resources, and use this information to design the infrastructure provision processes. Based on that, further estimations are derived e.g., the complexity, the initial and operational cost of the system or how long it takes to deploy it from scratch. This information is calculated by taking into account the resource requirements of the DARE services (e.g., Kubernetes setup [20], DB redundancy, abundance of Kubernetes nodes).

Initial infrastructure requirements should be chosen as conservatively as possible and, eventually, scale up to meet the increasing needs of a growing user base. Modesty on initial requirements will speed up and simplify deployment and maintenance. For instance, a minimal installation of the DARE platform may require one single node for the Kubernetes master which controls only two other nodes with a relatively small shared volume storage. Such a system cannot handle more than a few concurrent workflows, but it is perfect to demo the platform, scrutinize for functionality issues and run some components over moderately sized data sets. On other words, it is a proof that the platform works. More VMs and storage volumes can be reserved at a later step.

The aforementioned strategy is used to mitigate the total running cost, weather it is measured in resource consumption (for an academic or on premises system) or money (for a cloud vendor). To maximize resource utilization and minimize the cost, monitoring tools can be used to precisely describe resource usage. IaaS clouds feature some monitoring capabilities (e.g., Cloudwatch [21] for AWS, Telemetry [22] API for OpenStack), but custom tools like Prometheus, Grafana, Graphite, Kibana or InfluxDB can also be used, with the advantage of being able to monitor the other layers of the stack as well. If some of the reserved resources are maxed out (e.g., storage volumes), operators can incrementally increase only the lacking resources (e.g., add some storage to the volume pool). They can also reduce spending by pinning down underutilized resources and decrease them (e.g., destroy some Kubernetes nodes, if they are too many for the actual usage of the system).

2.2 Kubernetes Deployment

A Kubernetes deployment [7] consists of a kubernetes master, a key-value service (default: etcd) and the kubernetes workers. The master consists of a scheduler, a controller and an API and uses the key-value service to store various system states. Each worker consists of a container runtime (default: containerd), a communication agent (kube-proxy) and an internal controller (kubelet).
The simplest installation configuration is to deploy everything on one node. This setup suffers from obvious performance and scaling limitations, so it is suggested to deploy multiple worker nodes which interact with the master service over the K8s API. The key-value service can also be deployed at a separate node. Besides, the master and key-value services can be deployed redundantly on more than one node each.

Installing every kubernetes component manually is a challenging task (https://kubernetes.io/docs/setup/scratch/). Fortunately, there exist a few tools to assist or even automate the process. In all cases, infrastructure can be provisioned separately, but some of the tools we review can take care of this issue automatically. The following list is indicative of the most widely spread K8s setup applications and it is not a comprehensive catalog of all installation methods.

1. kubeadm [23] is a first-class citizen on the Kubernetes ecosystem. It is a secure and recommended way to bootstrap the Kubernetes cluster. It has a set of building blocks to setup the cluster and can be easily extended. It doesn't provision cloud resources.
2. Kubespray [24], an ansible-based utility, promises to setup highly available Kubernetes installations on clouds like AWS, GCE, Azure, OpenStack, or even on bare metal. Based on our tests, we conclude that it does not work on ~okeanos out of the box, due to minor differences

between the OpenStack API and Synnefo, which could be bypassed with well tested solutions like API mapping proxies or an extension of Kubespray.
3. Kops [25] can create, destroy, upgrade, and maintain production-grade, highly-available Kubernetes clusters from the command line. It can provision the machines as well, but is limited to AWS.
4. Minikube [26] is a good solution for small-scale (e.g., single node) deployments, useful for development and testing.

The performance of a kubernetes system is scaled up by increasing the number and capacity of worker nodes. If the system is setup with kubeadm, the additional VMs have to be created and provisioned manually so that they can be accessed by kubeadm through an IP and PPK. Kubeadm is then instructed to setup K8s worker software (including the container runtime) and inform the master about the new node.

2.2.1 Container runtimes

Kubernetes supports many container runtimes, like containerd [27], rkt [28] or lxd [29]. One of the most popular choices is Docker [30], which is not runtime but an engine built on top of containerd. Kubernetes has been proven to effectively orchestrate Docker containers.

Kubernetes can interact with any runtime compatible with the Container Runtime Interface [31]. CRI acts as a layer between kubelet and the runtime and it consists of an image service and a container service. The advantage of CRI is the ability to concurrently support different runtimes or container engines on the same K8s deployment. There are many implementations of CRI for K8s e.g., dockershim for docker support, cri-containerd to directly run components on containerd and cri-o [32] to run on OCI (Open Container Initiative) [33] compatible runtimes.

2.2.2 Network

To ensure a kubernetes cluster works, a unique IP must be assigned to each pod, containers or pods must be able to communicate with each other and an application must be able to communicate with the external world if configured.

For container networking, there are two specifications:

1. Container Network Model (CNM) [34], proposed by Docker
2. Container Network Interface (CNI) [35], proposed by CoreOS

Kubernetes uses CNI to assign IPs to each pod. The container runtime offloads the IP assignment to CNI, which connects to the underlying configured plugin, like Bridge or MACvlan, to get the IP address. Once the IP address is given by the respective plugin, CNI forwards it back to the requested container runtime. Therefore, a network bridge or other mechanism must be configured at an earlier step or available at the infrastructure layer.

With the help of the underlying host operating system, all of the container runtimes generally create an isolated network entity for each container that it starts. On Linux, that entity is referred to as a network namespace. These network namespaces can be shared across containers, or with the host operating system. Inside a Pod, containers share the network namespaces, so that they can reach to each other via localhost.

In a clustered environment, the Pods can be scheduled on any node. We need to make sure that the Pods can communicate across the nodes, and all the nodes should be able to reach any Pod. Kubernetes also puts a condition that there shouldn't be any Network Address Translation (NAT) while doing the Pod-to-Pod communication across hosts. We can achieve this via routable Pods and nodes (VMs) using the underlying physical infrastructure, or with Software Defined Network like Flannel [36] or Calico [37]. This is less complicated than it sounds, since most cloud vendors provide adequate networking features. For instance, VMs on ~okeanos can connect on a user defined private network with adequate support for routable Pods.

2.3 Storage

The DARE platform aspires to provide solutions on the field of big data, therefore storage is of paramount importance. Clouds typically offer both object and block storage and DARE can be benefited from both.

An object storage service is permanent and independent from any computational resources (no VMs necessary). Data are stored as objects rather than files on a namespaced environment which resembles the behavior of a filesystem. Some examples of object storage solutions are GRNET Pithos+ [8], Amazon S3 [9], Dropbox [38], Box, Google Drive. To access and object storage service an application must connect through a REST API, using a client provided by the service. For instance, kamaki python client is used to programmatically manage storage on Pithos+.

A block storage service is responsible for managing block devices (e.g., disk volumes) on a cloud environment. It is a substantial module for every computing service out there. Apart from accounting the disk space assigned to VMs, these services may support detaching and reattaching volumes between VMs. A more advanced but profoundly useful feature is the sharing of volumes, which requires an extra layer of storage management.

On the context on DARE, one should mention databases as a way to store structured, indexed data. On the cloud world, it is quite trivial to setup and deploy one or more DBMS e.g. using ansible scripts.

Container-based services like DARE require sharing data between applications (e.g., DARE components).

Object storage can be used to store and recover large bodies of data, which are then available to every service on the system, or to well defined scopes. Object storage services allow controlled access to data. For instance, on OpenStack Swift, stored objects are accessible only to a limited list of users, defined either one by one or in reusable groups. Access levels are distinguished in various levels:

ownership, read/write, read only. The DARE platform can take advantage of these features to define and implement data accessibility for large, raw data.
Implementing efficient sharing with block devices on a container-based system is a challenge with a lot of research going on. Some of the most reliable solutions follow:

1. NFS [39], a simple solution with low scaling threshold. On NFS, interconnected nodes get a copy of the accessed data over a network, thus increasing the number of nodes results to a non negligible decrease to I/O speed and reliability.
2. Ceph file system (CephFS) [40] is a POSIX-compliant filesystem that uses a Ceph Storage Cluster to store its data. The Ceph filesystem uses the same Ceph Storage Cluster system as Ceph Block Devices, Ceph Object Storage with its S3 and Swift APIs, or native bindings (librados). Using the Ceph Filesystem requires a Ceph Storage cluster and at least one Ceph Metadata Server setup in the Ceph Storage Cluster. Although Ceph clusters are offered as turnkey solutions by some vendors, significant installation and configuration is required to set it up on a plan IaaS. Ceph clusters scale well and can be used as both block and object storage solutions.
3. Amazon Elastic File System (EFS) [41] and Google Filestore [42] are similar solutions, which provide a simple filesystem interface over their massively scalable storage solutions. They both offer automations to use these systems along with Kubernetes instances, as long as they are deployed on an Amazon cloud (e.g., AWS) or on Google Compute Engine, respectively.
4. HDFS [43] or Hadoop file system is tailored to ensure high performance on operations over large, distributed data sets. HDFS is tightly related to the Hadoop system, and features a distributed, master-slave architecture. Installation and maintenance outside the scope of Hadoop is complex. Hadoop does not provide a POSIX-compatible interface, though, so it's not a fit for a general-purpose cluster.

As soon as the shared storage system is configured, using it with Kubernetes does not increase the installation complexity. Scaling is just a matter of increasing resources and updating the shared system accordingly, since K8s can automatically take advantage of the extra space.

2.4 Installing DARE components

The details and specifications of DARE components are not yet mature enough to justify a precise and comprehensive installation guide. DARE components should be containerized and able to run in K8s pods. There are two types of components:

- process elements (PEs), the instances of which are short lived processes which can tolerate failure and
- core components (e.g., databases, workflow engines, port endpoints) which are permanent and highly available.

Kubernetes explicitly supports both modes of operation.

It is not clear if components will necessarily make use of shared file systems or will interconnect otherwise (e.g., common object storage, network), but technologies are mature enough to support

both scenarios. Most PE instances are expected to run without external network access, that is, without an external IP, but internal networks must be available to allow interconnectivity and orchestration. This policy can me reconsidered, though, if PE requirements change (e.g., if some PEs require their instances to download information directly from external resources). Connectivity between PE instances and core components also relies on secure, internal networks handled by the container engine eachself.

3 Monitoring

3.1 Infrastructure

Most cloud systems provide statistics about actual resource usage (e.g., how much of each RAM a VM used or the network traffic over a channel), which are accessible through a UI in tabular or graphical format, or via a REST API. For instance, OpenStack-based clouds often implement Ceilometer, which collects and distributes usage information. AWS stores similar data as metadata on each cloud object and produces usage reports and custom alerts. The ~okeanos IaaS also stores usage information as metadata and provides some basic graphical information through the UI.

IaaS clouds also keep track of the amount of the reserved resources (e.g., how much RAM is assigned to a VM or the upper traffic limit of a network channel) and enforce limitations based on predefined quotas. ~okeanos implements a static model of hierarchical, fine grained quota management, where for every resource, a user or project account is charged exactly before the resource is reserved. AWS and Azure feature dynamically adjustable quotas, based on actual resource utilization and a set of user-defined rules triggered by alerts.

Infrastructure information can also be collected with monitoring tools like Prometheus [14], Grafana [15], Graphite [44], InfluxData [45] and ELK (Elasticsearch-Logstash-Kibana) [46]. A brief review of these tools follows in the next section, since they are mostly suitable to monitor the platform layer. Still, they are quite effective in monitoring the infrastructure as well. The benefit of this approach is that full stack statistics can be aggregated and exported over a single endpoint. DARE operators would find full-stack statistics very useful when tweaking the platform, while it also functions as an auxiliary debugging tool for the platform developers. Also, the REST APIs and plotting tools offered by dedicated monitoring services are usually better suited for full scale platform like DARE, compared to the ones provided by IaaS clouds.

3.2 Platform and services

The combination of logs and usage statistics on various layers of the DARE platform is the key to a smart monitoring and alerting plan which can provide rich in context and directly consumable information. We examine how every layer can be monitored and weather it is possible to use as few monitoring layers as possible for the whole stack.

Kubernetes is a complex super-service of paramount importance for the operational health of the DARE platform, thus it should be monitored. For this reason, the DARE platform should contain a monitoring and alerting tool. Although these tools can run in containers, it is important to run in isolation from the K8s cluster. This practice guarantees crucial monitoring and alerting services in case of K8s failures. Otherwise a collapse of K8s would bring down the monitoring services as well. This external service can also monitor the infrastructure, thus ripping the benefits described in the previous section.

For instance, a good monitoring setup for K8s may contain at least two Prometheus servers to scrape information from the nodes running K8s masters and provide queering and aggregation, a Grafana service to visualize Prometheus results in a meaningful way, and an Alert manager to notify operators on critical events.

The rest of the DARE components are deployed and managed as containers or pods inside the K8s cluster. Complete container statistics are collected by the container engine (e.g. Docker engine) as well as by the container platform (Kubernetes) and are exported over the K8s API.

As mentioned in previous sections, there are two types of DARE components: core (e.g., workflow engine) and PEs. They can be monitored by the same tools, but due to their differences (permanent and short lived). To tackle this issue, we adjust the frequency of statistic collection and the alerting policy to the needs of each type of component. For instance, the failure of a PE is not a crucial event, while the failure of a DB instance may render the whole system unusable.

All DARE monitoring needs for the components running inside the K8s cluster are adequately taken care off by K8s monitoring and alerting capabilities. K8s can monitor, log and restart services according to preset rules. The very concept of serving both permanent and short lived pods is a built-in feature of K8s. What's more, DARE developers can benefit from the ability to attach extra per-pod metadata, useful in building some core DARE services.

The only disadvantage of K8s as a monitoring and alerting tool is the low quality of graphic or tabular reports. Fortunately, tools like Grafana or Kibana can construct presentable graphs and aggregate many types of structured data, including K8s API responses.

3.3 Tools

Prometheus is a time series monitoring service, built with a server-client design. Prometheus agents are installed on the monitored nodes and configured to watch the services of interest. The agents "scrap" information which is pulled by the Prometheus server. The information is aggregated in order to provide arbitrary queries over a REST API. Prometheus installation may consists of one or more Prometheus servers for safety and makes use of DB for storing indices.

InfluxData or **influxDB** is the main competitor of Prometheus in time series monitoring services. It features a similar design with agents, aggregation and queries over a REST API. The main difference is that information is pushed by the agents rather than pulled by the server. Modification of polling intervals for various applications in InfluxDB requires a change on the node of each individual application, while on Prometheus this can be handled centrally on the server. There are also some differences in the way data is stored: InfluxDB uses a database for all purposes, while Prometheus uses it only for indices, as the rest of statistics are held in separate files.

Grafana is a monitoring and visualization tool. It can aggregate and visualize a long range of data and it can be extended with plugins. Although it could, alone, provide monitoring and alerting services, it is

usually combined with Prometheus or InfluxDB which perform better on large, concurrent statistics, but lack the intuitive presentation capabilities of Grafana.

Graphite is a data logging and graphing tool for time series data. For DARE purposes, it is similar to Grafana, since it is focused on storing and graphically representing data rather than collecting and aggregating them. Still, it does collect through a "push" mechanism. It does not support replication.

Another stack fit for the job is **ELK** or "**Elasticsearch-Logstash-Kibana**". Elasticsearch provides a data storage, aggregation and queering engine, Logstash collects the data and Kibana is responsible for visualizations. ELK is a general purpose search engine often used for monitoring. Elasticsearch processes data based on lexicographical and string operations, while Prometheus and InfluxDB are better for numerical comparisons.

4 Accounting and user management

4.1 Infrastructure

Infrastructure users are the people or organizations which setup and maintain the DARE stack, for instance administrators, operators or core DARE developers. They mustn't be confused with the users (e.g., scientists with big data workflows) or the final beneficiaries (e.g., third parties consuming the produced information) of the platform.

One approach is to maintain only one master account on the IaaS platform, which will be used to deploy and manage the application and its core services. This is not a personal, but an organizational account, access to which will be provided to the administrators and operators for as long as they work for the project. All resources are accounted to this single account. More elaborate accounting and billing needs are managed at a higher level.

Extending the aforementioned approach, there can be one account for each DARE stack deployed. For instance, there can be a development, a testing, a staging and a production deployment and a separate cloud account for each of them. Accessibility to these accounts must vary decreasingly, with the development stack being accessible by many people and production by a few trusted operators.

A fine grained infrastructure user policy consists of one account per person and a system that assigns different access rights to each account, regarding infrastructure resources. For instance, OpenStack features a hierarchical model of tenants, accounts and projects, where the owner and the tenant of the resource can be different, and an account can have controlled access to resources belonging to different accounts. Similarly, ~okeanos accounts can participate in one or more projects, which describe the ownership policy for all cloud resources and the quotas per user as well as per project. Cloud resource ownership can also be transferred to other accounts under the same project and multi-user access can be enabled with common SSH keys.

All major cloud vendors provide accounting and quota mechanisms to enforce restrictions and billing. AWS and Azure offer dynamic plans which can be adjusted as the system evolves. In ~okeanos, all resources are accounted (CPU, RAM, VMs, private networks, IPs, disk volumes, object storage), but they user has to be given permission in advance, through the projects mechanism. An ~okeanos project describes how a defined amount of resources can be distributed to the subscribed accounts. Each account can be subscribed in many projects and hold the ownership of the resources they reserve. If they release a resource, they stop being accountable for it. The resource is freed and reused by another cloud account.

4.2 Platform and services

There are at least three categories of users, based on the layer they are authenticated at. Operators and system experts authenticate on the first level (K8s master, reverse proxy, etc.) with PPK. Experts

and other contributors of data and processes authenticate against K8s, in order to configure workflows, deploy PEs, etc. Last but not least, visiting users can either be anonymous or authenticated on a higher level by the DARE portal. Operators of external services (e.g., deployment of K8s master) authenticate with PPK directly on the physical nodes. The other two categories are handled by K8s and by the DARE portal.

All Kubernetes clusters have two categories of users: service accounts managed by Kubernetes, and normal users.

Normal users are assumed to be managed by an outside, independent service like an admin distributing private keys, a user store like Keystone or Google Accounts, or a file with usernames and passwords. In this regard, Kubernetes does not have objects which represent normal user accounts. Normal users cannot be added to a cluster through an API call.

In contrast, service accounts are users managed by the Kubernetes API. They are bound to specific namespaces, and created automatically by the API server or manually through API calls. Service accounts are tied to a set of credentials stored as Secrets, which are mounted into pods allowing in-cluster processes to talk to the Kubernetes API.

API requests are tied to either a normal user or a service account, or are treated as anonymous requests. This means every process inside or outside the cluster must authenticate when making requests to the API server, or be treated as an anonymous user.

Kubernetes uses client certificates, bearer tokens, an authenticating proxy, or HTTP basic auth to authenticate API requests through authentication plugins. As HTTP requests are made to the API server, plugins attempt to associate the request with a username, a user uid or a group of users. Multiple authentication methods can be used at once. The system:authenticated group is included in the list of groups for all authenticated users.

Integrations with other authentication protocols (LDAP, SAML, Kerberos, alternate x509 schemes, etc) can be accomplished using an authenticating proxy or the authentication webhook.

4.3 PEs and platform users

Kubernetes user management suffices for controlling access on the core (permanent) DARE components, as well as PEs. The short lived tasks are deployed as a result of a series of decisions and operations performed by other DARE tools (e.g., a workflow manager). In that sense, they are systemic processes and should be owned by some K8s service accounts or scoped under specific namespaces.

On the other hand, PEs are related to (i.e., developed by, used by) functional users (not operators) who don't care about the complexity of the underlying system, but are relevant to the process execution in order to ensure accountability and ownership of the results. In this case, DARE design can benefit from the K8s ability to incorporate wide spread authentication protocols and mechanisms like Auth2 [47] or SAML/2 [48] and to scope users and APIs with fine grained access control. K8s can, in principle, be configured to authenticate academic users whose organizations are affiliated with an academic

federation (e.g., EOSChub [49], EduGAIN [50], Shibboleth [51], FedCloud [52]).

Kubernetes ships an integrated Role-Based Access Control (RBAC) [2] component that matches an incoming user or group to a set of permissions bundled into roles. These permissions combine verbs (get, create, delete) with resources (pods, services, nodes) and can be namespace or cluster scoped. A set of out of the box roles are provided that offer reasonable default separation of responsibility depending on what actions a client might want to perform. Simple and broad roles may be appropriate for smaller clusters, but in the context of DARE it may be useful to separate teams into separate namespaces with more limited roles.

Another approach is to delegate high level user management to custom tools built for that purpose in the context of DARE. The advantage of this approach is the implementation of a solution tailored to the needs of each kind of DARE user or participant. Still, such a system could be implemented as a layer on top of K8s user management system, thus making use of the user management features that are already functional, like authentication protocol support and pod access control. What's more, utilizing K8s user management can be beneficial when restring access to data produced and managed by a PE.

5 Security policies

5.1 Infrastructure

The security of a cloud infrastructure refers to the controlled access on the system and the protection of data and operations from accidental or deliberate damage. Strategies for increasing security involve minimizing the number of users who have access on a particular deployment, safeguarding VMs with PPK and other methods, and limiting the number of public exposure to nodes and services.

The access needs of every system are different. In case of DARE, there may be more than one deployments i.e., for development, testing, staging and production. The development stack must be accessible to a lot of different kinds of developers, devops, testers etc. in order to develop, deploy and test their code and artifacts. The integrity of the stack is based on development schemes like gitflow, to which everyone commits and respects, and can be supported by continuous integration tools like Jenkins, but no restriction mechanism are necessary. A testing environment is usually rebuilt as often as needed and it should be accessible by probably as many people as the development environment. Staging must be a stable environment, which is operated by a few people. Although staging is not exposed to the world, it must be protected from experimentation and untested features, otherwise it is no different to testing. Production should also be a stable installation of the system, with real, potentially sensitive data, which must be protected with controlled access. Variations of this model are quite common, for instance development and testing are sometimes merged into one stack.

Development and testing are experimental, disposable and often automatically built and destroyed by CI or orchestration tools. The data they process are either fake or insensitive (e.g., not real-time data loads but rather a snapshot of an old data set), so that if they leak or prove unprotected, the implications are negligible. This allows for a lower level of security. Access can be given to every expert or system developer assigned to explore options, develop features, test ideas or fix bugs on the platform.

On the other hand, staging and, especially, production should be accessible only by a few people with proven experience and knowledge on operating large systems, who are accountable for the well being of the platform. The number of these people must stay as low as possible, e.g., 2 or 3 persons. It is important to distinguish between experts who are responsible for the platform functionality and those who are responsible for running the platform and handling its real date, including sensitive user information.

As we focus on the system components, we must safeguard access to VMs. Large systems with multiple installations consist of multiple VMs. It is not uncommon some of these VMs to be created for a quick test and then forgotten alive for a long time, or to be deployed by creative experts who focus on functionality and optimizations rather than security. This is quite common at the development and testing stages.

Safeguarding VMs involves accessing them with safe methods like PPK (Private Public Key) rather than username/password. Large IaaS clouds provide key rings with user public keys. When a VM is created, the operator can select some of these keys, thus providing PPK access. On top, new VMs must be switch off username/password authentication and be disabled direct root access by allowing access to super users with sudo privileges. Last but not least, operators must install security updates as often as possible. This is a tedious task: a trivial operation must be repeated on a large number of VMs. Fortunately, it can be taken care of with low level automations like ansible or bash scripts.

From an infrastructure perspective, limiting access to services can be achieved by building private networks and minimizing connectivity to public networks as much as possible. Ideally, developers separate the services to be deployed with a public IP, from the services which can work fine on a private network. For instance, ~okeanos offers private networks that can be built from the UI, with dhcp and a predefined internal IP range. It is also easy to select which network each VM would connect to. Even VMs which must have some access to the internet, can operate on a private network with another VM acting as a proxy and a firewall. Some cloud providers like AWS offer turn-key proxies and firewalls as a service, for a price.

5.2 Platform and services

The Kubernetes stack (K8s API, scheduler, controller, etcd) along with its monitoring stack (e.g., Prometheus-based) constitute the group of first-level services, as opposed to the services running on higher levels, inside pods. First-level services are deployed on 3 or more VMs (without taking into account K8s nodes), assuming they are replicated and physically separated in order to increase availability.

In a secure configuration, services connect to the same network only if they have to exchange information. What's more, if a service does not require to be accessed from the outside world, it is not connected to a public network. Access to services can be further restricted to specific IPs or IP ranges through IPtables or Reverse proxies. They can also be protected by username/password on the HTTP level.

For example, we assume a stack of Prometheus-Grafana-Alertmanager to watch a K8s cluster. The Prometheus master scraps logs from the services through a common network. This network can be private, to secure the watched services. Grafana and Alert-manager must query Prometheus, but they don't need to be directly connected to K8s master or etcd. They also expose information to various roles, including to people with non-technical responsibilities, therefore they are connected to an external network, but protected by a reverse proxy with password and restricted to specific IPs. The monitoring stack components connect to each other through HTTPS and reverse proxies, or with a dedicated private network.

The Kubernetes stack must also run on a private network, which acts as an underlying network for all pods. Nodes connected on this network must be provided with a trusted certificate (e.g., a free "let's encrypt" certificate). Since all communication is implemented over HTTP, it is mandatory to use Transport Layer Security (TLS) for all API traffic. Note that some components and installation methods

may enable local ports over HTTP and administrators should familiarize themselves with the settings of each component to identify potentially insecure traffic.

Choosing an appropriate API and user authentication scheme contributes to improved security. API authentication can be based on Bearer tokens without requiring a user authentication. The later approach is sufficient for safe DARE components accessing each other or queering K8s for statistics on PE execution. Still, all API clients must be authenticated, even those that are part of the infrastructure like nodes, proxies, the scheduler, and volume plugins. These clients are typically service accounts or use x509 client certificates, and they are created automatically at cluster startup or are setup as part of the cluster installation.

Once authenticated, every API call is also expected to pass an authorization check. In the context of K8s, it is recommended to use the Node and RBAC authorizers together (see "Accounting and user management"). With authorization, it is important to understand how updates on one object may cause actions in other places. For instance, a user may not be able to create pods directly, but allowing them to create a deployment, which creates pods on their behalf, will let them create those pods indirectly. Likewise, deleting a node from the API will result in the pods scheduled to that node being terminated and recreated on other nodes. In the context of DARE, specific, custom roles should be carefully configured to prevent accidental escalation.

As mentioned in the "Accounting and user management" section, K8s supports various user authentication schemes e.g., LDAP, Auth2 or OIDC [53], so it is relatively straightforward to authenticate federated academic users in mass or selectively. In the context of DARE it is safer to restrict access to an approved list of experts and contributors. Handling arbitrary users can be delegated to the DARE proxy, which is more secure since it runs in an isolated environment (a pod).

5.3 Availability and Replication

The lower the platform layer, the more crucial it is to keep it available. First-level monitoring must be replicated and available in order to alert operators in case of a K8s failure. Kubernetes master components (scheduler, API, controller, etcd) must also be replicated, as the whole platform stands on their shoulders. This is ensured by taking good care of the monitoring system and establishing well defined procedures for operations.

Kubernetes ensures high availability of the core, permanent components, as long they are configured to be deployed as such. The platform maintains a history of states and frequent checks to decide if services run as expected. In case of a failure, the failed pod is killed and reloaded.

Replication is handled by the Replication Controller. If there are too many pods, the Replication Controller terminates the extra pods. If there are too few, the Replication Controller starts more. Unlike manually created pods, the pods maintained by a Replication Controller are automatically replaced if they fail, are deleted, or are terminated. For example, pods are re-created on a node after disruptive maintenance such as a kernel upgrade. For this reason, a Replication Controller is suggested even if the application require only a single pod. A Replication Controller is similar to a process supervisor, but

instead of supervising individual processes on a single node, it supervises multiple pods across multiple nodes. Replication Controller is often abbreviated to “rc” or “rcs” in discussion, and as a shortcut in kubectl commands.

A simple case is to create one Replication Controller object to reliably run one instance of a Pod indefinitely. A more complex use case is to run several identical replicas of a replicated service, such as web servers. In general, stateless services are easily replicated with K8s.

A less trivial case is that of a replicated database. Databases must be highly available but they are also stateful, so the various DB instances must be always in sync. Most popular DB systems can be configured to be replicated by advertising DBMS instances when they are loaded. Asynchronous replication performs better with reduced setup complexity. Synchronous replication ensures data persistence in case of a failure, but it slows down SQL commits and requires more cloud resources and a complicated setup.

5.4 PEs and pods

Process Elements are short lived components, launched automatically by workflow management tools through K8s and developed by various experts. Although security should always be a consideration, the focus of developing a process element is always its functionality and efficiency. Because of this, they are expected to cause various security issues.

The container/pod architecture ensures isolation of execution and the deployment of K8s cluster on a private network mitigate most security issues emanating from PEs. Still, malicious components could harm the infrastructure, the data or the other components if they are not restricted by a combination of Kubernetes configurations and human-directed policies.

PEs may have access to shared volumes, a permanent storage or the internet. Based on this fact, we can define scopes with different access rights. The most restricted scope should disable all access to storage or networks, but then the process would be useless. At the very minimum, the system needs a way to retrieve the process results e.g., by giving write access to a shared volume or providing access to permanent storage. DARE operators may define scopes for every useful use case (e.g., shared volume only, shared volume and external IP, etc.) and assign them to PEs according to their needs.

PEs can be audited by humans at a previous step. When a PE is submitted, it is considered a candidate. The developer must clearly state the requirements of the component regarding input and output methods and communication with other components or services. These requirements can be expressed as metadata attached on the component description. What's more, the developer may provide testing specifications, including a collection of test inputs with their corresponding expected outcomes. After the tester audits the source code, they would either deploy an on-premises stack or use the testing DARE installation to deploy the component and monitor its activity. Network monitoring tools can be used to spot suspicious operations emanating from the component and the outcome must be analyzed for malicious content. If the tests are passed, the component is released as safe and can be deployed on K8s, based on the aforementioned requirements.

6 Failure management

6.1 Infrastructure recovery

Infrastructure failures can be caused by natural or artificial factors and can compromise the efficiency or availability of resources. Cloud operators have developed elaborate techniques in order to manually or automatically recover faulty clusters or VMs by transferring the setup to different hardware. Disk failures are the most critical since they may cause data loss, although cloud vendors utilize replication techniques like RAID and frequent full backups.

Another method is the ability to quickly deploy using automation and orchestration tools like ansible. Maintaining automated deployment methods for the full stack is highly suggested for many reasons, one of which is quick recovery. Operators and developers must update the deployment scripts whenever the stack is modified. Still, this method does not backup or recover any data, it just builds a fresh stack.

Many cloud vendors allow snapshots of running VMs, which can then be downloaded or stored on safer, permanent storage repositories. Other vendors, like Amazon, have implemented a mechanism for incremental backups. In case of failure, VMs are restored to the snapshotted state, by loading the snapshots on healthy physical resources.

Administrators can apply similar methods to backup VMs incrementally with tools like duplicity or rsync. A combination of incremental backups with automation tools like ansible constitute an adequate method for full VM restoration. With duplicity, for example, one can backup only the parts of the system which are not recoverable by ansible (e.g., application and user data, secrets and passwords). These parts are compared with the last backup and the difference is encrypted and stored at a remote location e.g., an object storage service. In case of failure and loss of a VM, the operators execute the ansible scripts to rebuilt it and, using duplicity, they can restore it to the last saved state.

6.2 Recovering platform and services

Kubernetes is capable of recovering failed pods but it doesn't include an inherent mechanism for backing up and recovering configuration files, secrets, tokens and application data. This functionality is supported by some external tools like Heptio-Ark [54] and kube-backup [55].

Heptio-Ark uses git to store K8s configuration and it features various tools for backup and recovery. It also supports volume snapshots for some cloud vendors (AWS and Azure). To operate ark, a server instance has to be installed on the cluster, which can be managed from a local client.

Kube-backup export configured K8s resources using kubectl and pushes them to a git repository. Secrets can be dumped as well into the repository using git-crypt. However, it cannot snapshot persistent volumes.

If DARE is deployed on a cloud that doesn't support volume snapshots or is not supporter by ark, volumes must be backed up with explicit backup tools like the ones described in the previous subsection (e.g., duplicity). If all application data is backed up, the resources required to store the saved information would be unreasonably large. In general, it is a good idea to store persistent data to a permanent storage service and deploy on the cluster as many stateless components as possible. On the other hand, components like databases must be frequently dumped and incrementally and securely stored to a remote storage.

6.3 Storage and data

Storage backups are mandatory for failure recovery. On DARE platform there are many types of important data: the monitoring database, the K8s key-value store, the database with application metadata and the scientific datasets. This collection of data is different to the collection of credentials and configuration files used to restore services, since the later are handled by different recovering mechanisms, as described in the previous chapter.

Permanent storage services play a key role in securing and recovering large data backups. These services are parts of every major cloud offering (e.g., S3, Pithos+), but here are also stand-alone services which can be used (e.g., Dropbox). With tools like CEPH it is also possible to install and maintain in-premises object stores. All permanent storage solutions are based on mechanisms for retaining data even in case of failures on physical devices, like data replication and incremental modifications. What's more, operation control promises frequent large scale backups and recovery within an adjustable time frame. For instance, Dropbox guarantees 1 day window for data retainment on the free plan, but proposes various offerings for longer windows.

7 Conclusions

The DARE platform promises a feature-rich environment of interactions between various users, processes and data collections. In order to deliver, a multilayer architecture of services, network configurations and storage solutions is being designed. To deploy and maintain such an ambitious stack of services, we need tools to automate the process.

The key tool for the job is Kubernetes, a widely spread container orchestrator. Kubernetes is located between the infrastructure and the DARE services. What's more, user management, monitoring, workflow execution and collecting information for processes are going to be implemented based on APIs and features already provided by K8s.

We proposed methods for deploying Kubernetes on IaaS clouds, and discussed the pros and cons of coupling infrastructure provisioning with Kubernetes provisioning. Scaling issues were discussed from both perspectives: infrastructure and container platform. We focused on networking and shared volumes as potential challenges, and listed the most mature tools and services to deal with volume and object storage on a cloud-deployed K8s installation.

Monitoring was examined and industry standards were compared. We used the case of a Prometheus stack as an example to clarify specific operations and cases. Still, this guide does not decisively suggest a particular technology as the most suitable for DARE, since such a decision is out of the scope of this document. We did suggest, however, setting up an external monitoring stack, with the sole purpose of watching the container platform (K8s). The rest of the stack can be conveniently be monitored by K8s itself.

User management was discussed with regard to DARE requirements. A case was made that K8s supports elaborate user schemes and enforces access and restriction policies on both processes and data in a way that is suitable to the needs of DARE platform. These features are horizontal to the DARE stack, in the sense that they can be utilized by higher level services to implement various user schemes with complex interactions e.g., service operators, experts owning processes or data consumers. Apart from praising K8s for its user management abilities, we also mentioned authentication and authorization technologies that make sense in an academic context.

Our security guides deal with each layer separately. We suggest relying on well established policies and methods based on cryptography (PPK, TLS, HTTPS), obfuscation and access control (private networks, reverse proxies, mediated connections). Again, we praised K8s abilities to ensure some security features regarding core DARE services running as containers. What's more, we examined the case of auditing and executing potentially malicious PEs and policies for safeguarding the rest of the infrastructure.

The last chapter examined some failure recovery techniques. We focused on quick redeployments, database backups and stored data permanence. We presented tools for K8s backups and highlighted the importance of permanent storage services for retaining large data sets or backing up configurations and metadata.

To conclude, the DARE platform is an exciting challenge for operators and developers, because it combines a wide range of well established technologies in one stack. We generally suggest tested, mature technologies when building the infrastructure and platform, so that developers and engineers can alleviate their efforts on solving the issues of the higher DARE layers. Our guidelines are based on reviewing industry standards and they aim to support the DARE architecture as it was at the time this document was composed. As the DARE platform matures, policies and operations are due to adjustments or replacements.

8 References

[1] D4.7 and ID2.1-M8 Dare Architecture and Technical Positioning
[2] Kubernetes reference: https://kubernetes.io/docs/reference
[3] Docker swarm: https://docs.docker.com/engine/swarm/
[4] Apache Mesos: http://mesos.apache.org/
[5] Apache Marathon: https://mesosphere.github.io/marathon/
[6] Apache Chronos: https://mesos.github.io/chronos/
[7] Kubernetes concepts: https://kubernetes.io/docs/concepts/
[8] ~okeanos IaaS: https://okeanos.grnet.gr/home/
[9] Amazon AWS: https://aws.amazon.com/
[10] EOSC: https://www.eosc-portal.eu/
[11] Microsoft Azure: https://azure.microsoft.com
[12] Kamaki client: https://www.synnefo.org/docs/kamaki/latest
[13] Ansible: https://www.ansible.com/
[14] Prometheus: https://prometheus.io/
[15] Grafana: https://grafana.com/
[16] Puppet: https://puppet.com/
[17] Canonical Juju: https://jujucharms.com/
[18] OpenStack: https://www.openstack.org/
[19] Synnefo: https://www.synnefo.org/
[20] Kubernetes setup: https://kubernetes.io/docs/setup/
[21] Amazon Cloudwatch: https://aws.amazon.com/cloudwatch/
[22] OpenStack Telemetry: https://wiki.openstack.org/wiki/Telemetry
[23] Kubeadm: https://github.com/kubernetes/kubeadm
[24] Kubespray: https://github.com/kubernetes-incubator/kubespray
[25] Kops: https://github.com/kubernetes/kops
[26] Minikube: https://github.com/kubernetes/minikube
[27] Containerd: https://containerd.io
[28] RKT: https://coreos.com/rkt/
[29] LXD: https://linuxcontainers.org/lxd/
[30] Docker: https://www.docker.com/
[31] K8s CRI: https://github.com/kubernetes/community/blob/master/contributors/devel/container-runtime-interface.md
[32] CRI-O: https://cri-o.io/
[33] OCI: https://www.opencontainers.org/
[34] CNM: https://github.com/docker/libnetwork/blob/master/docs/design.md
[35] CNI: https://github.com/containernetworking/cni

[36] Flannel: https://coreos.com/blog/introducing-rudder.html
[37] Calico: https://www.projectcalico.org/
[38] Dropbox: https://www.dropbox.com/
[39] NFS: http://nfs.sourceforge.net/
[40] Ceph FS: http://docs.ceph.com/docs/master/cephfs/
[41] Amazon EFS: https://aws.amazon.com/efs/
[42] Google Filestore: https://cloud.google.com/filestore/
[43] Hadoop: https://hadoop.apache.org/
[44] Graphite: https://graphiteapp.org/
[45] InfluxDB or InfluxData: https://www.influxdata.com/
[46] Elastisearch-Logstash-Kibana: https://www.elastic.co/elk-stack
[47] Auth 2.0: https://oauth.net/2/
[48] SAML: http://saml.xml.org/
[49] EOSChub: https://eosc-hub.eu/events/eosc-hub-week-16-20-april-2018-malaga-spain/programme/eosc-aai
[50] EduGain: https://edugain.org/
[51] Shibboleth: https://www.shibboleth.net/
[52] Fedcloud: http://occi-wg.org/tag/fedcloud/
[53] OpenID Connect: https://openid.net/connect/
[54] Heptio Ark: https://github.com/heptio/ark
[55] Kube Backup: https://github.com/pieterlange/kube-backup