

METALK8S

MetalK8s Documentation

Release 2.4.3

Scality

Apr 17, 2020

CONTENTS:

I	Installation	1
1	Introduction	5
2	Prerequisites	13
3	Deployment of the Bootstrap node	17
4	Cluster expansion	21
5	Post-Installation Procedure	27
6	Accessing Cluster Services	31
II	Operational Guide	33
7	Bootstrap Node Backup and Restoration Procedure	37
8	Enable IP-in-IP encapsulation	39
9	ISO Preparation	41
10	Solutions Guide	43
11	Upgrade Guide	45
12	Downgrade Guide	47
13	Supported Versions	49
14	Downgrade Pre-requisites	51
15	Downgrade Steps	53
16	Changing the hostname of a MetalK8s node	55
17	Volume Management	57
18	Account Administration	65
19	Troubleshooting Guide	67
III	Developer Guide	71
20	Architecture Documents	73

21 Design Documents	99
22 How to build MetalK8s	107
23 How to run components locally	111
24 Development	115
25 Integrating with MetalK8s	127
 IV Glossary	 137
Index	141

Part I

Installation

This guide describes how to set up a [MetalK8s](#) cluster. It offers general requirements and describes sizing, configuration, and deployment. It also explains major concepts central to MetalK8s architecture, and shows how to access various services after completing the setup.

INTRODUCTION

1.1 Foreword

MetalK8s is a [Kubernetes](#) distribution with a number of add-ons selected for on-premises deployments, including pre-configured monitoring and alerting, self-healing system configuration, and more.

The installation of a MetalK8s cluster can be broken down into the following steps:

1. *Setup* of the environment
2. *Deployment* of the *Bootstrap node*, the first machine in the cluster
3. *Expansion* of the cluster, orchestrated from the Bootstrap node
4. *Post installation* configuration steps and sanity checks

1.2 Choosing a Deployment Architecture

Before starting the installation, choosing an architecture is recommended, as it can impact sizing of the machines and other infrastructure-related details.

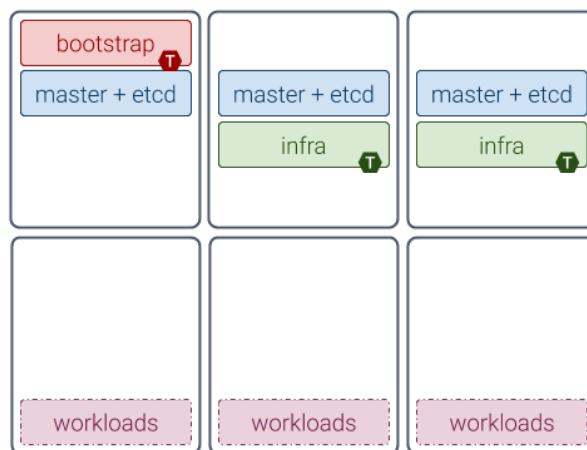
Note: “Machines” may indicate bare-metal servers or VMs interchangeably.

Warning: MetalK8s is not designed to handle world-distributed multi-sites architectures. Instead, it focuses on providing a highly resilient cluster at the datacenter scale. To manage multiple sites, look into solutions provided at the application level, or alternatives from the community (such as what [the SIG Multicluster](#) provides).

1.2.1 Standard Architecture

The recommended architecture when installing a small-sized MetalK8s cluster emphasizes ease of installation, while providing a high stability for the scheduled workloads:

- One machine running Bootstrap and control plane services
- Two other machines running control plane and Infra services
- Three more machines for workload applications

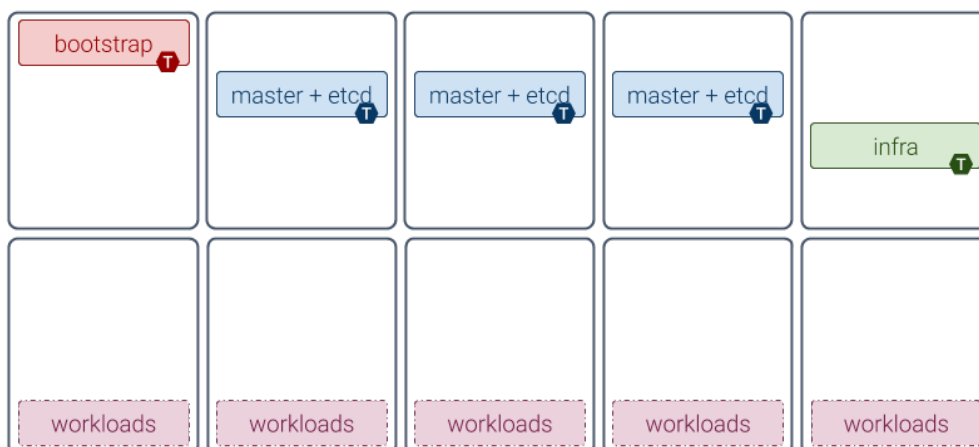


Machines dedicated to the control plane do not need large amounts of resources (see [the sizing notes below](#)), and can safely run as virtual machines. Running workloads on dedicated machines also allows for simpler sizing of said machines, as MetalK8s impact would be negligible.

1.2.2 Extended Architecture

This example architecture focuses on reliability rather than compactness offering the finest control over the entire platform:

- One machine dedicated to running Bootstrap services (see [the Bootstrap role](#) definition below)
- Three extra machines (or five if installing a really large cluster, e.g. >100 nodes) for running the [Kubernetes](#) control plane (with [core K8s services](#) and the backing [etcd DB](#))
- One or more machines dedicated to running Infra services (see [the Infra role](#))
- Any number of machines dedicated to running applications, the number and [sizing](#) depending on the applications (for instance, [Zenko](#) would recommend using three or more machines)

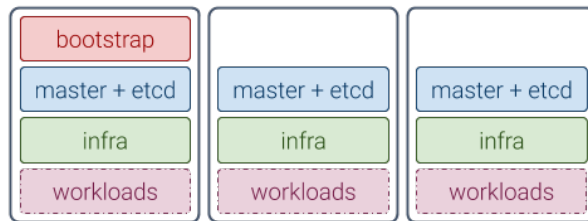


1.2.3 Compact Architectures

While not being focused on having the smallest compute and memory footprints, MetalK8s can provide a fully functional single node “cluster”. The Bootstrap node can be configured to also allow running applications next to all the other services required (see [the section about taints](#) below).

A single node cluster does not provide any form of resilience to machine or site failure, which is why the recommended most compact architecture to use in production includes three machines:

- Two machines running control plane services alongside infra and workload applications
- One machine running Bootstrap services in addition to all the other services



Note: Sizing of such compact clusters needs to account for the expected load, and the exact impact of colocating an application with MetalK8s services needs to be evaluated by said application’s provider.

1.2.4 Variations

It is possible to customize the chosen architecture using combinations of [roles](#) and [taints](#), which are described below, to adapt to the available infrastructure.

As a general recommendation, it is easier to monitor and operate well-isolated groups of machines in the cluster, where hardware issues would only impact one group of services.

It is also possible to evolve an architecture after initial deployment, in case the underlying infrastructure also evolves (new machines can be added through the [expansion](#) mechanism, roles can be added or removed...).

1.3 Concepts

Although being familiar with [Kubernetes concepts](#) is recommended, the necessary concepts to grasp before installing a MetalK8s cluster are presented here.

1.3.1 Nodes

[Nodes](#) are Kubernetes worker machines, which allow running containers and can be managed by the cluster (control plane services, described below).

1.3.2 Control Plane and Workload Plane

This dichotomy is central to MetalK8s, and often referred to in other Kubernetes concepts.

The **control plane** is the set of machines (called *nodes*) and the services running there that make up the essential Kubernetes functionality for running containerized applications, managing declarative objects, and providing authentication/authorization to end-users as well as services. The main components making up a Kubernetes control plane are:

- *API Server*
- *Scheduler*
- *Controller Manager*

The **workload plane** indicates the set of nodes where applications will be deployed via Kubernetes objects, managed by services provided by the **control plane**.

Note: Nodes may belong to both planes, so that one can run applications alongside the control plane services.

Control plane nodes often are responsible for providing storage for *API Server*, by running *etcd*. This responsibility may be offloaded to other nodes from the workload plane (without the *etcd* taint).

1.3.3 Node Roles

Determining a *Node* responsibilities is achieved using **roles**. Roles are stored in *Node manifests* using labels, of the form `node-role.kubernetes.io/<role-name>: ''`.

MetalK8s uses five different **roles**, that may be combined freely:

node-role.kubernetes.io/master The master role marks a control plane member. control plane services (see above) can only be scheduled on master nodes.

node-role.kubernetes.io/etcd The *etcd* role marks a node running *etcd* for storage of *API Server*.

node-role.kubernetes.io/infra The *infra* role is specific to MetalK8s. It serves for marking nodes where non-critical services provided by the cluster (monitoring stack, UIs, etc.) are running.

node-role.kubernetes.io/bootstrap This marks the *Bootstrap node*. This node is unique in the cluster, and is solely responsible for the following services:

- An RPM package repository used by cluster members
- An OCI registry for *Pods* images
- A *Salt Master* and its associated *SaltAPI*

In practice, this role is used in conjunction with the *master* and *etcd* roles for bootstrapping the control plane.

In the *architecture diagrams* presented above, each box represents a role (with the `node-role.kubernetes.io/` prefix omitted).

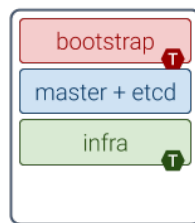
1.3.4 Node Taints

Taints are complementary to roles. When a taint or a set of taints is applied to a *Node*, only *Pods* with the corresponding *tolerations* can be scheduled on that Node.

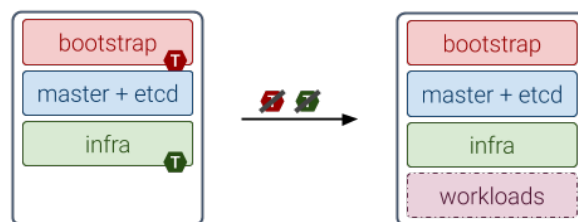
Taints allow dedicating Nodes to specific use-cases, such as having Nodes dedicated to running control plane services.

Refer to the *architecture diagrams* above for examples: each **T** marker on a role means the taint corresponding to this role has been applied on the Node.

Note that Pods from the control plane services (corresponding to master and etcd roles) have tolerations for the bootstrap and infra taints. This is because after *bootstrapping the first Node*, it will be configured as follows:



The taints applied are only tolerated by services deployed by MetalK8s. If the selected architecture requires workloads to run on the Bootstrap node, these taints should be removed.



To achieve this, use the following commands after deployment:

```
root@bootstrap $ kubectl taint nodes <bootstrap-node-name> \
    node-role.kubernetes.io/bootstrap:NoSchedule-
root@bootstrap $ kubectl taint nodes <bootstrap-node-name> \
    node-role.kubernetes.io/infra:NoSchedule-
```

Note: To get more in-depth information about taints and tolerations, see [the official Kubernetes documentation](#).

1.3.5 Networks

A MetalK8s cluster requires a physical network for both the control plane and the workload plane Nodes. Although these may be the same network, the distinction will still be made in further references to these networks, and when referring to a Node IP address. Each Node in the cluster **must** belong to these two networks.

The control plane network will serve for cluster services to communicate with each other. The workload plane network will serve for exposing applications, including the ones in infra Nodes, to the outside world.

Todo: Reference Ingress

MetalK8s also allows one to configure virtual networks used for internal communications:

- A network for [Pods](#), defaulting to 10.233.0.0/16
- A network for [Services](#), defaulting to 10.96.0.0/12

In case of conflicts with the existing infrastructure, make sure to choose other ranges during the [Bootstrap configuration](#).

1.4 Additional Notes

1.4.1 Sizing

Defining an appropriate sizing for the machines in a MetalK8s cluster strongly depends on the selected architecture and the expected future variations to this architecture. Refer to the documentation of the applications planned to run in the deployed cluster before completing the sizing, as their needs will compete with the cluster's.

Each [role](#), describing a group of services, requires a certain amount of resources for it to run properly. If multiple roles are used on a single Node, these requirements add up.

Role	Services	CPU	RAM	Required storage	Recom- mended storage
bootstrap	Package repositories, container registries, Salt master	1 core	2 GB	Sufficient space for the prod- uct ISO archives	
etcd	etcd database for K8s API	0.5 core	1 GB	1 GB for /var/lib/etcd	
master	K8s API, scheduler, and controllers	0.5 core	1 GB		
infra	Monitoring services, Ingress controllers	0.5 core	2 GB	10 GB partition for Prometheus 1 GB parti- tion for Alertmanager	
<i>requirements common to any Node</i>	Salt minion, Kubelet	0.2 core	0.5 GB	40 GB root partition	100 GB or more for / var

These numbers are not accounting for highly unstable workloads or other sources of unpredictable load on the cluster services, and it is recommended to provide an additional 50% of resources as a safety margin.

Consider the [official recommendations for etcd sizing](#) as the stability of a MetalK8s installation depends strongly on the backing etcd stability (see [this note](#) for more details). Prometheus and Alertmanager also require storage, as explained in [this section](#).

1.4.2 Deploying with Cloud Providers

When installing in a virtual environment, such as [AWS EC2](#) or [OpenStack](#), special care will be needed for adjusting networks configuration. Virtual environments often add a layer of security at the port level, which should be disabled, or circumvented with *[IP-in-IP encapsulation](#)*.

Also note that Kubernetes has numerous integrations with existing cloud providers to provide easier access to proprietary features, such as load balancers. For more information, see [this documentation article](#).

PREREQUISITES

[MetalK8s](#) clusters require machines running [CentOS](#) / [RHEL](#) 7.6 or higher as their operating system. These machines may be virtual or physical, with no difference in setup procedure. The number of machines to setup depends on the chosen architecture (see [Choosing a Deployment Architecture](#)).

Machines must **not** be managed by any configuration management system (e.g. [SaltStack](#), [Puppet](#)).

Warning: Distribution must be, as much as possible, left intact (no tuning, tweaking, configuration nor software installation).

2.1 Proxies

For nodes operating behind a proxy, see [Configuration](#)

2.2 Linux Kernel Version

Linux Kernel shipped with latest (7.7) and previous versions of [CentOS](#) / [RHEL](#) 7 is affected by a cgroups memory leak bug.

Kernel must be at least in version 3.10.0-1062.4.1 for this bug to be fixed.

The version can be retrieved using:

```
uname -r
```

If the installed version is lower than the one above, it must be upgraded:

```
yum upgrade -y kernel-3.10.0-1062.4.1.el7
reboot
```

2.3 Provisioning

2.3.1 SSH

Each machine must be accessible through SSH from the host. As part of the [Deployment of the Bootstrap node](#), a new SSH identity for the [Bootstrap node](#) will be generated and shared to other nodes in the cluster. It is also possible to do it beforehand.

2.3.2 Network

Each machine must be a member of both the control plane and workload plane networks, as described in [Networks](#). However, these networks can overlap, and nodes need not have distinct IPs for each plane.

For the host to reach the cluster-provided UIs, it must be able to connect to control plane IPs of the machines.

2.3.3 Repositories

Each machine needs to have repositories properly configured and having access to basic repository packages (depending on the operating systems).

CentOS:

- base
- extras
- updates

RHEL:

- rhel-7-server-rpms
- rhel-7-server-extras-rpms
- rhel-7-server-optional-rpms

Note: For RHEL you should have [a system properly registered](#).

Note: The repository names and configurations do not necessarily need to be the same as the official ones but all packages must be made available.

Enable an existing repository:

CentOS:

```
yum-config-manager --enable <repo_name>
```

RHEL:

```
subscription-manager repos --enable=<repo_name>
```

Add a new repository:

```
yum-config-manager --add-repo <repo_url>
```

Note: *repo_url* can be remote url using prefix *http://*, *https://*, *ftp://*, ... or a local path using *file://*.

For more detail(s), refer to the official Red Hat documentation:

- [Enable Optional repositories with RHSM](#)
- [Configure repositories with YUM](#)
- [Advanced repositories configuration](#)

2.3.4 etcd

For production environments, a dedicated block device for etcd is recommended for better performance and stability. If possible, use a SSD which provides lower write latencies, with less variance than a spinning disk, thus improving the reliability of etcd.

The device must be formatted and mounted on `/var/lib/etcd`, on Nodes intended to bear the *etcd role*.

For further information on etcd hardware requirements, see the [official documentation](#).

DEPLOYMENT OF THE BOOTSTRAP NODE

3.1 Preparation

1. Build the ISO using [this procedure](#). Scality customers can retrieve validated builds as part of their license from the Scality repositories.
2. Download the MetalK8s ISO file on the machine that will host the bootstrap node. Mount this ISO file at the specific following path:

```
root@bootstrap $ mkdir -p /srv/scality/metalk8s-2.4.3
root@bootstrap $ mount <path-to-iso> /srv/scality/metalk8s-2.4.3
```

3.2 Configuration

1. Create the MetalK8s configuration directory.

```
root@bootstrap $ mkdir /etc/metalk8s
```

2. Create the `/etc/metalk8s/bootstrap.yaml` file. This file contains initial configuration settings which are mandatory for setting up a MetalK8s [Bootstrap node](#). Change the networks, IP address, and hostname fields to conform to your infrastructure.

```
apiVersion: metalk8s.scality.com/v1alpha2
kind: BootstrapConfiguration
networks:
  controlPlane: <CIDR-notation>
  workloadPlane: <CIDR-notation>
  pods: <CIDR-notation>
  services: <CIDR-notation>
proxies:
  http: <http://proxy-ip:proxy-port>
  https: <https://proxy-ip:proxy-port>
  no_proxy:
    - <host>
    - <ip/cidr>
ca:
  minion: <hostname-of-the-bootstrap-node>
archives:
  - <path-to-metalk8s-iso>
```

The `networks` field specifies a range of IP addresses written in CIDR notation for its various subfields.

The `controlPlane` and `workloadPlane` entries are **mandatory**. These values specify the range of IP addresses that will be used at the host level for each member of the cluster.

```
networks:
  controlPlane: 10.200.1.0/28
  workloadPlane: 10.200.1.0/28
```

All nodes within the cluster **must** connect to both the control plane and workload plane networks. If the same network range is chosen for both the control plane and workload plane networks then the same interface may be used.

The pods and services fields are not mandatory, though can be changed to match the constraints of existing networking infrastructure (for example, if all or part of these default subnets is already routed). During installation, by default pods and services are set to the following values below if omitted.

For **production clusters**, we advise users to anticipate future expansions and use sufficiently large networks for pods and services.

```
networks:
  pods: 10.233.0.0/16
  services: 10.96.0.0/12
```

The proxies field can be omitted if there is no proxy to configure. The 2 entries http and https are used to configure the containerd daemon proxy to fetch extra container images from outside the MetalK8s cluster. The no_proxy entry specifies IPs that should be excluded from proxying, it must be a list of hosts, IP addresses or IP ranges in CIDR format. For example;

```
no_proxy:
- localhost
- 127.0.0.1
- 10.10.0.0/16
- 192.168.0.0/16
```

The archives field is a list of absolute paths to MetalK8s ISO files. When the bootstrap script is executed, those ISOs are automatically mounted and the system is configured to re-mount them automatically after a reboot.

3.3 SSH Provisioning

1. Prepare the MetalK8s PKI directory.

```
root@bootstrap $ mkdir -p /etc/metalk8s/pki
```

2. Generate a passwordless SSH key that will be used for authentication to future new nodes.

```
root@bootstrap $ ssh-keygen -t rsa -b 4096 -N '' -f /etc/metalk8s/pki/salt-bootstrap
```

Warning: Although the key name is not critical (will be re-used afterwards, so make sure to replace occurrences of salt-bootstrap where relevant), this key must exist in the /etc/metalk8s/pki directory.

3. Accept the new identity on future new nodes (run from your host).

1. Retrieve the public key from the Bootstrap node.

```
user@host $ scp root@bootstrap:/etc/metalk8s/pki/salt-bootstrap.pub /tmp/salt-bootstrap.
↵ pub
```

2. Authorize this public key on each new node (this command assumes a functional SSH access from your host to the target node). Repeat until all nodes accept SSH connections from the Bootstrap node.

```
user@host $ ssh-copy-id -i /tmp/salt-bootstrap.pub root@<node_hostname>
```

3.4 Installation

3.4.1 Run the Installation

Run the bootstrap script to install binaries and services required on the Bootstrap node.

```
root@bootstrap $ /srv/scality/metalk8s-2.4.3/bootstrap.sh
```

Warning: For virtual networks (or any network which enforces source and destination fields of IP packets to correspond to the MAC address(es)), *IP-in-IP needs to be enabled*.

3.4.2 Validate the install

- Check that all *Pods* on the Bootstrap node are in the **Running** state. Note that Prometheus and Alertmanager pods will remain in a **Pending** state until their respective persistent storage volumes are provisioned.

Note: The administrator *kubeconfig* file is used to configure access to Kubernetes when used with *kubectl* as shown below. This file contains sensitive information and should be kept securely.

On all subsequent *kubectl* commands, you may omit the `--kubeconfig` argument if you have exported the KUBECONFIG environment variable set to the path of the administrator *kubeconfig* file for the cluster.

By default, this path is `/etc/kubernetes/admin.conf`.

```
root@bootstrap $ export KUBECONFIG=/etc/kubernetes/admin.conf
```

```
root@bootstrap $ kubectl get nodes --kubeconfig /etc/kubernetes/admin.conf
```

NAME	STATUS	ROLES	AGE	VERSION
bootstrap	Ready	bootstrap,etcd,infra,master	17m	v1.15.5


```
root@bootstrap $ kubectl get pods --all-namespaces -o wide --kubeconfig /etc/kubernetes/admin.conf
```

NAMESPACE	NAME	IP	NODE	NOMINATED NODE	READY	STATUS	
↪RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES		
kube-system	6m29s	10.233.220.129	bootstrap	<none>	1/1	Running	0_
↪							
kube-system	6m29s	10.200.3.152	bootstrap	<none>	1/1	Running	0_
↪							
kube-system	6m29s	10.233.220.134	bootstrap	<none>	1/1	Running	0_
↪							
kube-system	6m29s	10.233.220.132	bootstrap	<none>	1/1	Running	0_
↪							
kube-system	5m45s	10.200.3.152	bootstrap	<none>	1/1	Running	0_
↪							
kube-system	5m57s	10.200.3.152	bootstrap	<none>	1/1	Running	0_
↪							
kube-system	7m4s	10.200.3.152	bootstrap	<none>	1/1	Running	0_
↪							
kube-system	6m32s	10.200.3.152	bootstrap	<none>	1/1	Running	0_

(continues on next page)

(continued from previous page)

kube-system	kube-scheduler-bootstrap	1/1	Running	0_
↪ 7m4s	10.200.3.152 bootstrap <none> <none>			
kube-system	repositories-bootstrap	1/1	Running	0_
↪ 6m20s	10.200.3.152 bootstrap <none> <none>			
kube-system	salt-master-bootstrap	2/2	Running	0_
↪ 6m10s	10.200.3.152 bootstrap <none> <none>			
kube-system	storage-operator-7567748b6d-hp7gq	1/1	Running	0_
↪ 6m6s	10.233.220.138 bootstrap <none> <none>			
metalk8s-ingress	nginx-ingress-control-plane-controller-5nkkx	1/1	Running	0_
↪ 6m6s	10.233.220.137 bootstrap <none> <none>			
metalk8s-ingress	nginx-ingress-controller-shg7x	1/1	Running	0_
↪ 6m7s	10.233.220.135 bootstrap <none> <none>			
metalk8s-ingress	nginx-ingress-default-backend-7d8898655c-jj7l6	1/1	Running	0_
↪ 6m7s	10.233.220.136 bootstrap <none> <none>			
metalk8s-monitoring	alertmanager-prometheus-operator-alertmanager-0	0/2	Pending	0_
↪ 6m1s	<none> <none> <none> <none>			
metalk8s-monitoring	prometheus-operator-grafana-775fbb5b-sgng	2/2	Running	0_
↪ 6m17s	10.233.220.130 bootstrap <none> <none>			
metalk8s-monitoring	prometheus-operator-kube-state-metrics-7587b4897c-tt79q	1/1	Running	0_
↪ 6m17s	10.233.220.131 bootstrap <none> <none>			
metalk8s-monitoring	prometheus-operator-operator-7446d89644-zqdlj	1/1	Running	0_
↪ 6m17s	10.233.220.133 bootstrap <none> <none>			
metalk8s-monitoring	prometheus-operator-prometheus-node-exporter-rb969	1/1	Running	0_
↪ 6m17s	10.200.3.152 bootstrap <none> <none>			
metalk8s-monitoring	prometheus-prometheus-operator-prometheus-0	0/3	Pending	0_
↪ 5m50s	<none> <none> <none> <none>			
metalk8s-ui	metalk8s-ui-6f74ff4bc-fgk86	1/1	Running	0_
↪ 6m4s	10.233.220.139 bootstrap <none> <none>			

- From the console output above, *Prometheus* and *Alertmanager* pods are in a Pending state because their respective persistent storage volumes need to be provisioned. To provision these persistent storage volumes, follow [this procedure](#).
- Check that you can access the MetalK8s GUI after the *installation* is completed by following [this procedure](#).
- At this stage, the MetalK8s GUI should be up and ready for you to explore.

Note: Monitoring through the MetalK8s GUI will not be available until persistent storage volumes for both Prometheus and Alertmanager have been successfully provisioned.

- If you encounter an error during installation or have difficulties validating a fresh MetalK8s installation, visit our [Troubleshooting guide](#).

CLUSTER EXPANSION

Once the *Bootstrap node* has been installed (see *Deployment of the Bootstrap node*), the cluster can be expanded. Unlike the `kubeadm join` approach which relies on *bootstrap tokens* and manual operations on each node, MetalK8s uses Salt SSH to setup new *Nodes* through declarative configuration, from a single endpoint. This operation can be done either through *the MetalK8s GUI* or *the command-line*.

4.1 Defining an Architecture

Follow the recommendations provided in *the introduction* to choose an architecture.

List the machines to deploy and their associated roles, and deploy each of them using the following process, either from *the GUI* or *CLI*. Note however, that the finest control over *roles* and *taints* can only be achieved using the command-line.

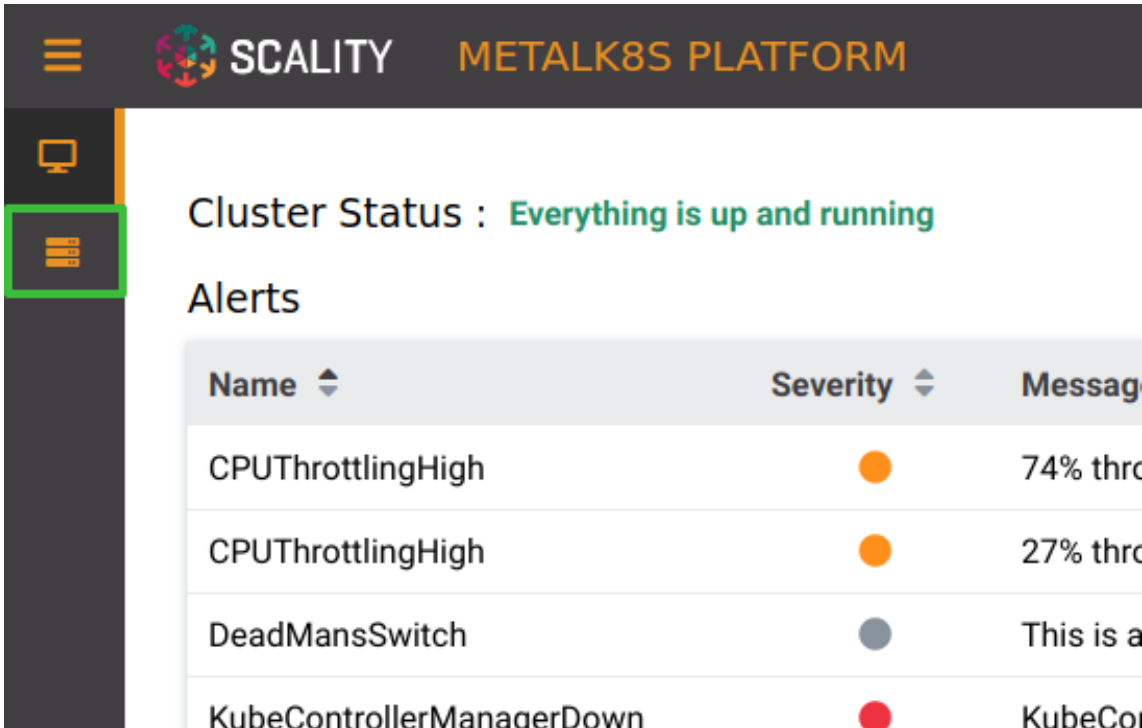
4.2 Adding a Node with the MetalK8s GUI

To reach the UI, refer to *this procedure*.

4.2.1 Creating a Node Object

The first step to adding a Node to a cluster is to declare it in the API. The MetalK8s GUI provides a simple form for that purpose.

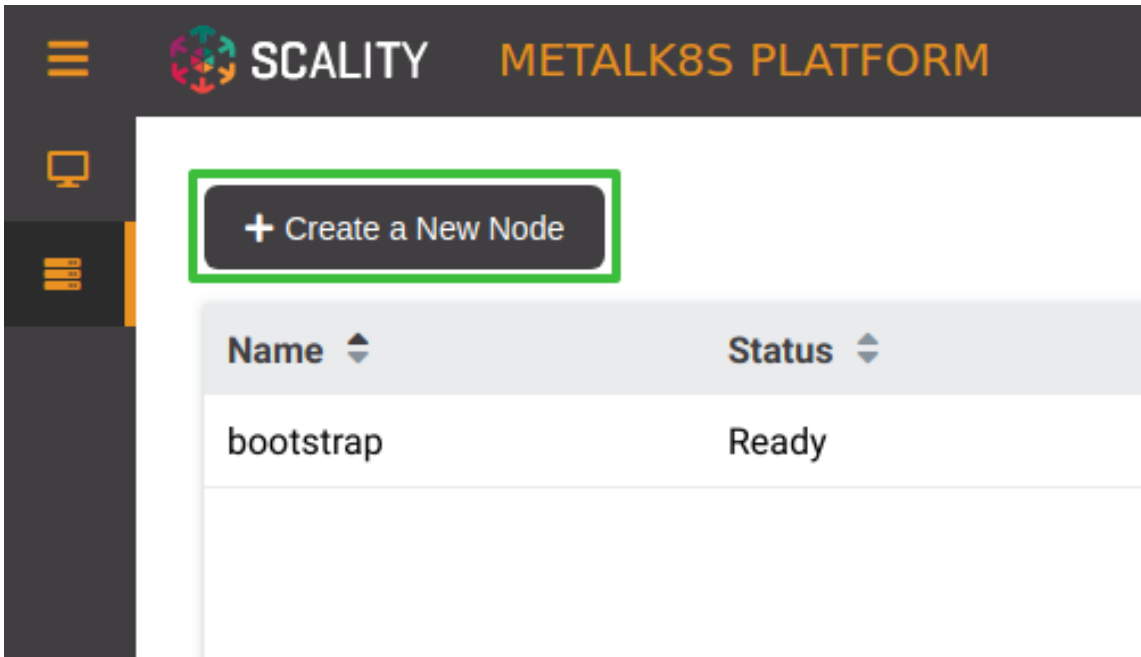
1. Navigate to the Node list page, by clicking the button in the sidebar:



The screenshot shows the SCALITY METALK8S PLATFORM dashboard. The top navigation bar includes the SCALITY logo and the text "METALK8S PLATFORM". On the left sidebar, the "Alerts" icon is highlighted with a green box. The main content area displays "Cluster Status : Everything is up and running" in green text. Below this, the "Alerts" section contains a table with the following data:

Name	Severity	Message
CPUThrottlingHigh	Orange circle	74% thrc
CPUThrottlingHigh	Orange circle	27% thrc
DeadMansSwitch	Grey circle	This is a
KubeControllerManagerDown	Red circle	KubeCo

- From the Node list (the Bootstrap node should be visible there), click the button labeled “Create a New Node”:



The screenshot shows the SCALITY METALK8S PLATFORM dashboard. The top navigation bar includes the SCALITY logo and the text "METALK8S PLATFORM". On the left sidebar, the "Nodes" icon is highlighted with a green box. The main content area displays a button labeled "+ Create a New Node" which is also highlighted with a green box. Below the button, a table shows the following data:

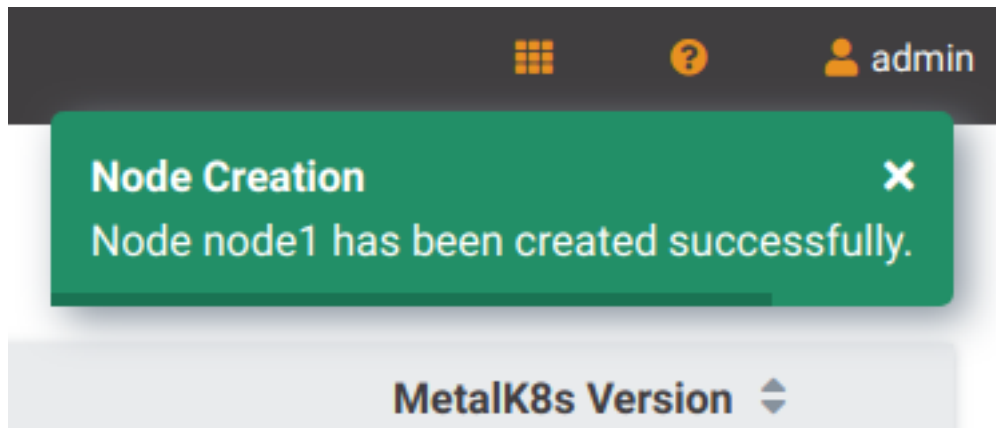
Name	Status
bootstrap	Ready

- Fill the form with relevant information (make sure the *SSH provisioning* for the Bootstrap node is done first):
 - Name:** the hostname of the new Node
 - SSH User:** the user for which the Bootstrap has SSH access
 - Hostname or IP:** the address to use for SSH from the Bootstrap
 - SSH Port:** the port to use for SSH from the Bootstrap
 - SSH Key Path:** the path to the private key generated in *this procedure*
 - Sudo required:** whether the SSH deployment will need sudo access

- **Roles/Workload Plane:** enable any workload applications run on this Node
- **Roles/Control Plane:** enable master and etcd services run on this Node
- **Roles/Infra:** enable infra services run on this Node

Note: Combination of multiple roles is possible: Selecting **Workload Plane** and **Infra** checkbox will result in infra services and workload applications run on this Node.

4. Click **Create**. You will be redirected to the Node list page, and will be shown a notification to confirm the Node creation:



4.2.2 Deploying the Node

After the desired state has been declared, it can be applied to the machine. The MetalK8s GUI uses [SaltAPI](#) to orchestrate the deployment.

1. From the Node list page, click the **Deploy** button for any Node that has not yet been deployed.

Name ▾	Status ▾	Deployment ▾
bootstrap	Ready	
node1	Unknown	Deploy

Once clicked, the button changes to **Deploying**. Click it again to open the deployment status page:



Detailed events are shown on the right of this page, for advanced users to debug in case of errors.

Todo:

- UI should parse these events further
 - Events should be documented
-

2. When deployment is complete, click **Back to nodes list**. The new Node should be in a **Ready** state.

Todo:

- troubleshooting (example errors)
-

4.3 Adding a Node from the Command-line

4.3.1 Creating a Manifest

Adding a node requires the creation of a *manifest* file, following the template below:

```
apiVersion: v1
kind: Node
metadata:
  name: <node_name>
  annotations:
    metalk8s.scality.com/ssh-key-path: /etc/metalk8s/pki/salt-bootstrap
    metalk8s.scality.com/ssh-host: <node control plane IP>
    metalk8s.scality.com/ssh-sudo: 'false'
  labels:
    metalk8s.scality.com/version: '2.4.3'
    <role labels>
spec:
  taints: <taints>
```

The combination of <role labels> and <taints> will determine what is installed and deployed on the Node.

roles determine a Node responsibilities. *taints* are complementary to roles.

- A node exclusively in the control plane with etcd storage
roles and taints both are set to master and etcd. It has the same behavior as the **Control Plane** checkbox in the GUI.

```
[...]
metadata:
  [...]
  labels:
    node-role.kubernetes.io/master: ''
    node-role.kubernetes.io/etcd: ''
    [...] (other labels except roles)
spec:
  [...]
  taints:
  - effect: NoSchedule
    key: node-role.kubernetes.io/master
  - effect: NoSchedule
    key: node-role.kubernetes.io/etcd
```

- A worker node dedicated to infra services (see *Introduction*)

roles and taints both are set to infra. It has the same behavior as the **Infra** checkbox in the GUI.

```
[...]
metadata:
  [...]
  labels:
    node-role.kubernetes.io/infra: ''
    [...] (other labels except roles)
spec:
  [...]
  taints:
  - effect: NoSchedule
    key: node-role.kubernetes.io/infra
```

- A simple worker still accepting infra services would use the same role label without the taint roles are set to node and infra. It's the same as the checkbox of Workload Plane and Infra in MetalK8s GUI.

4.3.2 CLI-only actions

- A Node dedicated to etcd
roles and taints both are set to etcd.

```
[...]
metadata:
  [...]
  labels:
    node-role.kubernetes.io/etcd: ''
    [...] (other labels except roles)
spec:
  [...]
  taints:
  - effect: NoSchedule
    key: node-role.kubernetes.io/etcd
```

4.3.3 Creating the Node Object

Use kubectl to send the manifest file created before to Kubernetes API.

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf apply -f <path-to-node-manifest>
node/<node-name> created
```

Check that it is available in the API and has the expected roles.

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf get nodes
NAME                STATUS    ROLES                  AGE      VERSION
bootstrap           Ready     bootstrap,etcd,infra, 12d      v1.11.7
<node-name>         Unknown   <expected node roles> 29s
```

4.3.4 Deploying the Node

Open a terminal in the Salt Master container using *this procedure*.

1. Check that SSH access from the Salt Master to the new node is properly configured (see *SSH Provisioning*).

```
root@salt-master-bootstrap $ salt-ssh --roster kubernetes <node-name> test.ping
<node-name>:
  True
```

2. Start the node deployment.

```
root@salt-master-bootstrap $ salt-run state.orchestrate metalk8s.orchestrate.deploy_
↪node \
                                saltenv=metalk8s-2.4.3 \
                                pillar='{ "orchestrate": { "node_name": "<node-name>" } }'

... lots of output ...
Summary for bootstrap_master
-----
Succeeded: 7 (changed=7)
Failed:    0
-----
Total states run:      7
Total run time: 121.468 s
```

Todo: Troubleshooting section

- explain orchestrate output and how to find errors
 - point to log files
-

4.4 Checking Cluster Health

During the expansion, it is recommended to check the cluster state between each node addition.

When expanding the control plane, one can check the etcd cluster health:

```
root@bootstrap $ kubectl -n kube-system exec -ti etcd-bootstrap sh --kubeconfig /etc/kubernetes/
↪admin.conf
root@etcd-bootstrap $ etcdctl --endpoints=https://[127.0.0.1]:2379 \
                        --ca-file=/etc/kubernetes/pki/etcd/ca.crt \
                        --cert-file=/etc/kubernetes/pki/etcd/healthcheck-client.crt \
                        --key-file=/etc/kubernetes/pki/etcd/healthcheck-client.key \
                        cluster-health

member 46af28ca4af6c465 is healthy: got healthy result from https://172.21.254.6:2379
member 81de403db853107e is healthy: got healthy result from https://172.21.254.7:2379
member 8878627efe0f46be is healthy: got healthy result from https://172.21.254.8:2379
cluster is healthy
```

Todo:

- add sanity checks for Pods lists (also in the relevant sections in services)
-

POST-INSTALLATION PROCEDURE

5.1 Provision Storage for Prometheus Services

After bootstrapping the cluster, the Prometheus and AlertManager services used to monitor the system **will not be running** (the respective *Pods* will remain in *Pending* state), because they require persistent storage to be available.

You can either provision these storage volumes on the *Bootstrap node*, or later on other nodes joining the cluster. It is even recommended to separate *Bootstrap services* from *Infra services*.

To create the required *Volume* objects, write a YAML file with the following contents, replacing `<node_name>` with the name of the *Node* on which to run Prometheus and AlertManager, and `<device_path[2]>` with the `/dev` path for the partitions to use:

```
---
apiVersion: storage.metalk8s.scality.com/v1alpha1
kind: Volume
metadata:
  name: <node_name>-prometheus
spec:
  nodeName: <node_name>
  storageClassName: metalk8s-prometheus
  rawBlockDevice: # Choose a device with at least 10GiB capacity
    devicePath: <device_path>
  template:
    metadata:
      labels:
        app.kubernetes.io/name: 'prometheus-operator-prometheus'
---
apiVersion: storage.metalk8s.scality.com/v1alpha1
kind: Volume
metadata:
  name: <node_name>-alertmanager
spec:
  nodeName: <node_name>
  storageClassName: metalk8s-prometheus
  rawBlockDevice: # Choose a device with at least 1GiB capacity
    devicePath: <device_path2>
  template:
    metadata:
      labels:
        app.kubernetes.io/name: 'prometheus-operator-alertmanager'
---
```

Once this file is created with the right values filled in, run the following command to create the *Volume* objects (replacing `<file_path>` with the path of the aforementioned YAML file):

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
  apply -f <file_path>
```

For more details on the available options for storage management, see [this section of the Operational Guide](#).

Todo:

- Sanity check
 - Troubleshooting if needed
-

5.2 Changing credentials

After a fresh installation, an administrator account is created with default credentials. For production deployments, make sure to change those credentials and use safer values.

To change user credentials and groups for *K8s API* (and as such, for *MetalK8s GUI* and *SaltAPI*), follow [this procedure](#).

To change Grafana user credentials, follow [this procedure](#).

5.3 Validating the deployment

To ensure the Kubernetes cluster is properly running before scheduling applications, perform the following sanity checks:

1. Check that all desired Nodes are in a **Ready** state and show the expected *roles*:

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
    get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
bootstrap	Ready	bootstrap,etcd,infra,master	42m	v1.15.5
node-1	Ready	etcd,infra,master	26m	v1.15.5
node-2	Ready	etcd,infra,master	25m	v1.15.5

Use the `kubectl describe node <node_name>` to get more details about a Node (for instance, to check the right *taints* are applied).

2. Check that *Pods* are in their expected state (most of the time, **Running**, except for Prometheus and AlertManager if the required storage was not provisioned yet - see [the procedure above](#)).

To look for all Pods at once, use the `--all-namespaces` flag. On the other hand, use the `-n` or `--namespace` option to select Pods in a given *Namespace*.

For instance, to check all Pods making up the cluster-critical services:

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
    get pods --namespace kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
apiserver-proxy-bootstrap	1/1	Running	0	43m
apiserver-proxy-node-1	1/1	Running	0	2m28s
apiserver-proxy-node-2	1/1	Running	0	9m
calico-kube-controllers-6d8db9bcf5-w5w94	1/1	Running	0	43m
calico-node-4vxpp	1/1	Running	0	43m
calico-node-hvlkx	1/1	Running	7	23m
calico-node-jhj4r	1/1	Running	0	8m59s
coredns-8576b4bf99-lfjfc	1/1	Running	0	43m
coredns-8576b4bf99-tnt6b	1/1	Running	0	43m
etcd-bootstrap	1/1	Running	0	43m
etcd-node-1	1/1	Running	0	3m47s

(continues on next page)

(continued from previous page)

etcd-node-2	1/1	Running	3	8m58s
kube-apiserver-bootstrap	1/1	Running	0	43m
kube-apiserver-node-1	1/1	Running	0	2m45s
kube-apiserver-node-2	1/1	Running	0	7m31s
kube-controller-manager-bootstrap	1/1	Running	3	44m
kube-controller-manager-node-1	1/1	Running	1	2m39s
kube-controller-manager-node-2	1/1	Running	2	7m25s
kube-proxy-gnxtp	1/1	Running	0	28m
kube-proxy-kvtjm	1/1	Running	0	43m
kube-proxy-vggzg	1/1	Running	0	27m
kube-scheduler-bootstrap	1/1	Running	1	44m
kube-scheduler-node-1	1/1	Running	0	2m39s
kube-scheduler-node-2	1/1	Running	0	7m25s
repositories-bootstrap	1/1	Running	0	44m
salt-master-bootstrap	2/2	Running	0	44m
storage-operator-756b87c78f-mjqc5	1/1	Running	1	43m

- Using the result of the above command, obtain a shell in a running etcd Pod (replacing <etcd_pod_name> with the appropriate value):

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
    exec --namespace kube-system -it <etcd_pod_name> sh
```

Once in this shell, use the following to obtain health information for the etcd cluster:

```
root@etcd-bootstrap $ etcdctl --endpoints=https://[127.0.0.1]:2379 \
    --ca-file=/etc/kubernetes/pki/etcd/ca.crt \
    --cert-file=/etc/kubernetes/pki/etcd/healthcheck-client.crt \
    --key-file=/etc/kubernetes/pki/etcd/healthcheck-client.key \
    cluster-health

member 46af28ca4af6c465 is healthy: got healthy result from https://<first-node-ip>:2379
member 81de403db853107e is healthy: got healthy result from https://<second-node-ip>:2379
member 8878627efe0f46be is healthy: got healthy result from https://<third-node-ip>:2379
cluster is healthy
```

- Finally, check that the exposed services are accessible, using the information from [this document](#).

ACCESSING CLUSTER SERVICES

6.1 MetalK8s GUI

This GUI is deployed during the *Bootstrap installation*, and can be used for operating, extending and upgrading a MetalK8s cluster.

6.1.1 Gather Required Information

Get the control plane IP of the bootstrap node.

```
root@bootstrap $ salt-call grains.get metalk8s:control_plane_ip
local:
  <the control plane IP>
```

6.1.2 Use MetalK8s UI

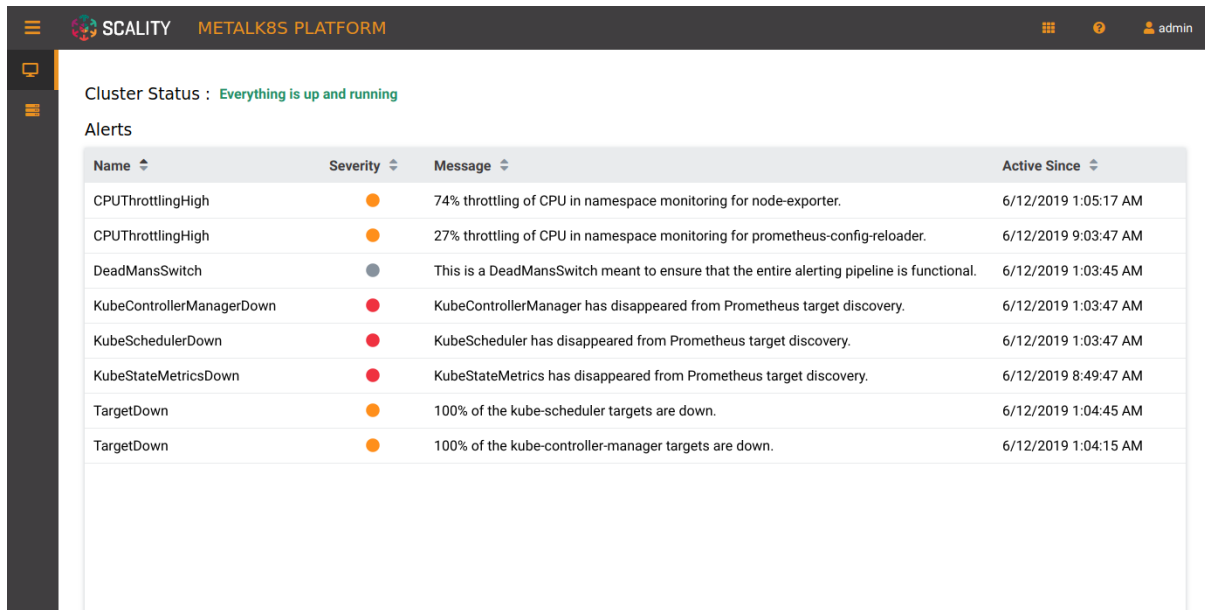
Once you have gathered the IP address and the port number, open your web browser and navigate to the URL `https://<ip>:8443`, replacing placeholders with the values retrieved before.

The login page is loaded, and should resemble the following:



Log in with the default login / password (admin / admin).

The landing page should look like this:



This page displays two monitoring indicators:

1. the Cluster Status, which evaluates if control plane services are all up and running
2. the list of alerts stored in [Alertmanager](#).

6.2 Grafana

Grafana is available on the same host as the MetalK8s UI, under /grafana. Log in with the default credentials: admin / admin.

6.3 Salt

MetalK8s uses [SaltStack](#) to manage the cluster. The Salt Master runs in a [Pod](#) on the [Bootstrap node](#).

The Pod name is salt-master-<bootstrap hostname>, and it contains two containers: salt-master and salt-api.

To interact with the Salt Master with the usual CLIs, open a terminal in the salt-master container (assuming the Bootstrap hostname to be bootstrap):

```
root@bootstrap $ kubectl exec -it -n kube-system -c salt-master --kubeconfig /etc/kubernetes/admin.conf salt-master-bootstrap bash
```

Todo:

- how to access / use SaltAPI
 - how to get logs from these containers
-

Part II

Operational Guide

This guide describes MetalK8s ISO preparation steps, upgrade and downgrade guidelines, supported versions and best practices required for operating [MetalK8s](#). Refer to the [Installation](#) if you do not have a working [MetalK8s](#) setup.

BOOTSTRAP NODE BACKUP AND RESTORATION PROCEDURE

This section describes how to backup a **MetalK8s** bootstrap node and how to restore a bootstrap node from such backup.

7.1 Backup procedure

A backup file is generated at the end of the bootstrap.

To create a new backup file you can run the following command:

```
/srv/scality/metalk8s-X.X.X/backup.sh
```

Backup archives are stored in `/var/lib/metalk8s/`.

7.2 Restoration procedure

Warning: It is mandatory to have a highly available control plane, with at least 3 members in the etcd cluster (including the failed bootstrap Node), to use the restore script.

Before running the script, the unreachable etcd member needs to be unregistered from the cluster. To do so, run the following commands from a working Node with the etcd role:

```
# Get etcd container id
CONT_ID=$(crictl ps -q --label io.kubernetes.container.name=etcd --state Running)

# List all etcd members to get the ID of the etcd member that need to be removed
crictl exec -it "$CONT_ID" \
  etcdctl --endpoints https://localhost:2379 \
  --ca-file /etc/kubernetes/pki/etcd/ca.crt \
  --key-file /etc/kubernetes/pki/etcd/server.key \
  --cert-file /etc/kubernetes/pki/etcd/server.crt \
  member list

# Remove the etcd member (replace <etcd_id> in the command)
crictl exec -it "$CONT_ID" \
  etcdctl --endpoints https://localhost:2379 \
  --ca-file /etc/kubernetes/pki/etcd/ca.crt \
  --key-file /etc/kubernetes/pki/etcd/server.key \
  --cert-file /etc/kubernetes/pki/etcd/server.crt \
  member remove <etcd_id>
```

To restore a bootstrap node you need a backup archive and **MetalK8s** ISOs.

All the ISOs referenced in the bootstrap configuration file (located at `/etc/metalk8s/bootstrap.yaml`) must be present.

First mount the ISO and then run the restore script:

```
/srv/scality/metalk8s-X.X.X/restore.sh --backup-file <backup_archive> --apiserver-node-ip <node_ip>
```

Note: Replace `<backup_archive>` with the path to the backup archive you want to use and `<node_ip>` with a control-plane IP of one control-plane Node.

ENABLE IP-IN-IP ENCAPSULATION

By default [Calico](#) in MetalK8s is configured to use [IP-in-IP](#) encapsulation only for cross-subnet communication.

[IP-in-IP](#) is needed for any network which enforces source and destination fields of IP packets to correspond to the MAC address(es).

To always use [IP-in-IP](#) encapsulation run the following command:

```
$ kubectl --kubeconfig /etc/kubernetes/admin.conf \
  patch ippool default-ipv4-ippool --type=merge \
  --patch '{"spec": {"ipipMode": "Always"}}'
```

For more details refer to [IP-in-IP Calico configuration](#).

ISO PREPARATION

This section describes a reliable way for provisioning a new **Metalk8s** ISO for upgrade or downgrade.

To provision a new **Metalk8s** ISO you need to run the utility script shipped with the current installation:

```
/srv/scality/metalk8s-X.X.X/iso-manager.sh -a <path_to_iso>
```


SOLUTIONS GUIDE

To deploy a Solution in a MetalK8s cluster, a utility script is provided. This section describes, step by step, how to deploy a Solution using this tool, located at the root of MetalK8s archive:

```
/srv/scality/metalk8s-2.4.3/solutions.sh
```

10.1 Import a Solution

First, the Solution must be imported in the cluster (make the container images available through the cluster registry):

```
./solutions.sh import --archive </path/to/solution.iso>
```

10.2 Activate a Solution Version

Only one version of a Solution can be active at any point in time. An active Solution version provides the cluster-wide resources, such as CRDs, to all other versions of this Solution. To activate a version, run:

```
./solutions.sh activate --name <solution-name> --version <solution-version>
```

10.3 Environment Creation

Solutions are meant to be deployed in isolated namespaces, which we call Environments. To create an Environment, run:

```
./solutions.sh create-env --name <environment-name>
```

10.4 Adding a Solution Version to an Environment

Solutions are packaged with an Operator, and optionally an associated web UI, to provide all required domain-specific logic. To deploy a Solution Operator and its UI in an Environment, run:

```
./solutions.sh add-solution --name <environment-name> \  
--solution <solution-name> --version <solution-version>
```

10.5 Configure a Solution

The Solution Operator and UI (if any) are now deployed. To finalize deployment and configuration of the Solution application, please refer to its documentation.

UPGRADE GUIDE

Upgrading a MetalK8s cluster is handled via utility scripts which are packaged with every new release. This section describes a reliable upgrade procedure for **MetalK8s** including all the components that are included in the stack.

11.1 Supported Versions

Note: MetalK8 supports upgrade **strictly** from one supported minor version to another. For example:

- Upgrade from 2.0.x to 2.0.x
- Upgrade from 2.0.x to 2.1.x

Please refer to the [release notes](#) for more information.

11.2 Upgrade Pre-requisites

Before proceeding with the upgrade procedure, make sure to complete the pre-requisites listed in *ISO Preparation*.

11.2.1 Run pre-check

You can test if your environment will successfully upgrade with the following command. This will simulate the upgrade prechecks and provide an overview of the changes to be carried out in your MetalK8s cluster.

Important:

The version prefix `metalk8s-X.X.X` as used below during a MetalK8s upgrade must be the new MetalK8s version you would like to upgrade to.

```
/srv/scality/metalk8s-X.X.X/upgrade.sh --destination-version \  
<destination_version> --dry-run --verbose
```

11.2.2 Backup old credentials

Starting 2.5.0, MetalK8s will henceforth implement OpenID Connect (OIDC) based authentication. Both K8s and Grafana will be configured to make use of the same OIDC provider.

Warning: Before running an upgrade from 2.4.x to 2.5.0 or higher, MetalK8s administrators **must** ensure all static users defined in `/etc/kubernetes/htpasswd` can be recreated, and if any, all users that were defined in Grafana. The upgrade procedure will result in all admin credentials being reset to their default values, and any additional user being removed. MetalK8s administrators need to remember and reconfigure these username/password pairs.

After upgrade is complete, a procedure for configuring the OIDC provider (Dex) user store will be provided in the next version.

11.3 Upgrade Steps

Ensure that the upgrade pre-requisites above have been met before you make any step further.

To upgrade a MetalK8s cluster, run the utility script shipped with the **new** version you want to upgrade to providing it with the destination version:

Important: The version prefix `metalk8s-X.X.X` as used below during a MetalK8s upgrade must be the new MetalK8s version you would like to upgrade to.

- From the *Bootstrap node*, launch the upgrade.

```
/srv/scality/metalk8s-X.X.X/upgrade.sh --destination-version <destination_version>
```

DOWNGRADE GUIDE

Downgrading a MetalK8s cluster is handled via utility scripts which are packaged with your current installation. This section describes a reliable downgrade procedure for **MetalK8s** including all the components that are included in the stack.

SUPPORTED VERSIONS

Note: MetalK8 supports downgrade **strictly** from one supported minor version to another. For example:

- Downgrade from 2.1.x to 2.0.x
- Downgrade from 2.2.x to 2.1.x

Please refer to the [release notes](#) for more information.

DOWNGRADE PRE-REQUISITES

Before proceeding with the downgrade procedure, make sure to complete the pre-requisites listed in *ISO Preparation*.

14.1 Run pre-check

You can test if your environment will successfully downgrade with the following command. This will simulate the downgrade prechecks and provide an overview of the changes to be carried out in your MetalK8s cluster.

Important:

The version prefix `metalk8s-X.X.X` as used below during a MetalK8s downgrade must be the currently-installed MetalK8s version.

```
/srv/scality/metalk8s-X.X.X/downgrade.sh --destination-version \  
<destination_version> --dry-run --verbose
```

DOWNGRADE STEPS

Ensure that the downgrade pre-requisites above have been met before you make any step further.

To downgrade a MetalK8s cluster, run the utility script shipped with the **current** installation providing it with the destination version:

Important: The version prefix `metalk8s-X.X.X` as used below during a MetalK8s downgrade must be the currently-installed MetalKs8 version.

- From the *Bootstrap node*, launch the downgrade.

```
/srv/scality/metalk8s-X.X.X/downgrade.sh --destination-version <version>
```


CHANGING THE HOSTNAME OF A METALK8S NODE

1. On the node, change the hostname:

```
$ hostnamectl set-hostname <New hostname>
$ systemctl restart systemd-hostnamed
```

2. Check that the change is taken into account.

```
$ hostnamectl status

Static hostname: <New hostname>
Pretty hostname: <New hostname>
Icon name: computer-vm
Chassis: vm
Machine ID: 5003025f93c1a84914ea5ae66519c100
Boot ID: f28d5c64f06c48a3a775e24c4f03d00c
Virtualization: kvm
Operating System: CentOS Linux 7 (Core)
CPE OS Name: cpe:/o:centos:centos:7
Kernel: Linux 3.10.0-957.12.2.el7.x86_64
Architecture: x86-64
```

3. On the bootstrap node, check the hostname edition incurred a change of status on the bootstrap. The edited node must be in a **NotReady** status.

```
$ kubectl get <node_name>
<node_name>    NotReady    etcd,master    19h    v1.11.7
```

4. Change the name of the node in the yaml file used to create it. Refer to [Creating a Manifest](#) for more information.

```
apiVersion: v1
kind: Node
metadata:
  name: <New_node_name>
  annotations:
    metalk8s.scality.com/ssh-key-path: /etc/metalk8s/pki/salt-bootstrap
    metalk8s.scality.com/ssh-host: <node control-plane IP>
    metalk8s.scality.com/ssh-sudo: 'false'
  labels:
    metalk8s.scality.com/version: '2.4.3'
    <role labels>
spec:
  taints: <taints>
```

Then apply the configuration:

```
$ kubectl apply -f <path to edited manifest>
```

5. Delete the old node (here <node_name>):

```
$ kubectl delete node <node_name>
```

6. Open a terminal into the *Salt master* container:

```
$ kubectl -it exec salt-master-<bootstrap_node_name> -n kube-system -c salt-master bash
```

7. Delete the now obsolete *Salt minion* key for the changed Node:

```
$ salt-key -d <node_name>
```

8. Re-run the deployment for the edited Node:

```
$ salt-run state.orchestrate metalk8s.orchestrate.deploy_node saltenv=metalk8s-2.4.  
→3 pillar='{ "orchestrate": { "node_name": "<new-node-name>" } }'
```

```
Summary for bootstrap_master
```

```
-----
```

```
Succeeded: 11 (changed=9)
```

```
Failed:    0
```

```
-----
```

```
Total states run:    11
```

```
Total run time: 132.435 s
```

9. On the edited node, restart the *kubelet* service:

```
$ systemctl restart kubelet
```

VOLUME MANAGEMENT

This section highlights **MetalK8s Volume Management** which covers volume creation and volume deletion necessary for use in persistent data storage within a MetalK8s Cluster.

17.1 StorageClass Creation

MetalK8s uses **StorageClass** objects to describe how **Volumes** are formatted and mounted. This section highlights how to create a Storageclass using the **CLI**.

1. Create a **StorageClass** manifest.

You can define a new **StorageClass** using the following template:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: <storageclass_name>
provisioner: kubernetes.io/no-provisioner
reclaimPolicy: Retain
volumeBindingMode: WaitForFirstConsumer
mountOptions:
  - rw
parameters:
  fsType: <filesystem_type>
  mkfsOptions: <mkfs_options>
```

Set the following fields:

- **mountOptions**: specifies how the volume should be mounted. For example **rw** (read/write), **ro** (read-only).
- **fsType**: specifies the filesystem to use on the volume. **xfs** and **ext4** are the only currently supported file system types.
- **mkfsOptions**: specifies how the volume should be formatted. This field is optional (note that the options are passed as a JSON-encoded string). For example **["-m", "0"]** could be used as **mkfsOptions** for an **ext4** volume.
- Set **volumeBindingMode** as **WaitForFirstConsumer** in order to delay the binding and provisioning of a Pod until a Pod using the **PersistentVolumeClaim** is created.

2. Create the **StorageClass**.

```
root@bootstrap $ kubectl apply -f storageclass.yml
```

3. Check that the **StorageClass** has been created.

```
root@bootstrap $ kubectl get storageclass <storageclass_name>
NAME                                PROVISIONER                AGE
<storageclass_name>               kubernetes.io/no-provisioner  2s
```

17.2 Volume Management using the CLI

To use persistent storage in a MetalK8s cluster, one needs to create **Volume** objects. In order to create Volumes you need to have **StorageClass** objects registered in your cluster. See [StorageClass Creation](#)

17.2.1 Volume Creation

This section describes how to create a **Volume** from the **CLI**.

1. Create a **Volume** manifest

You can define a new **Volume** using the following template:

```
apiVersion: storage.metal8s.scality.com/v1alpha1
kind: Volume
metadata:
  name: <volume_name>
spec:
  nodeName: <node_name>
  storageClassName: <storageclass_name>
  rawBlockDevice:
    devicePath: <device_path>
```

Set the following fields:

- **name**: the name of your volume, must be unique
- **nodeName**: the name of the node where the volume will be located.
- **storageClassName**: the **StorageClass** to use
- **devicePath**: path to the block device (for example, `/dev/sda1`).

2. Create the **Volume**

```
root@bootstrap $ kubectl apply -f volume.yml
```

3. Verify that the **Volume** was created

```
root@bootstrap $ kubectl get volume <volume_name>
NAME          NODE          STORAGECLASS
<volume_name> bootstrap    metal8s-demo-storageclass
```

17.2.2 Volume Deletion

This section highlights how to delete a **Volume** in a MetalK8s cluster using the **CLI**

1. Delete a **Volume**

```
root@bootstrap $ kubectl delete volume <volume_name>
volume.storage.metal8s.scality.com <volume_name> deleted
```

2. Check that the **Volume** has been deleted

Note: The command below returns a list of all volumes. The deleted volume entry should not be found in the list.

```
root@bootstrap $ kubectl get volume
```

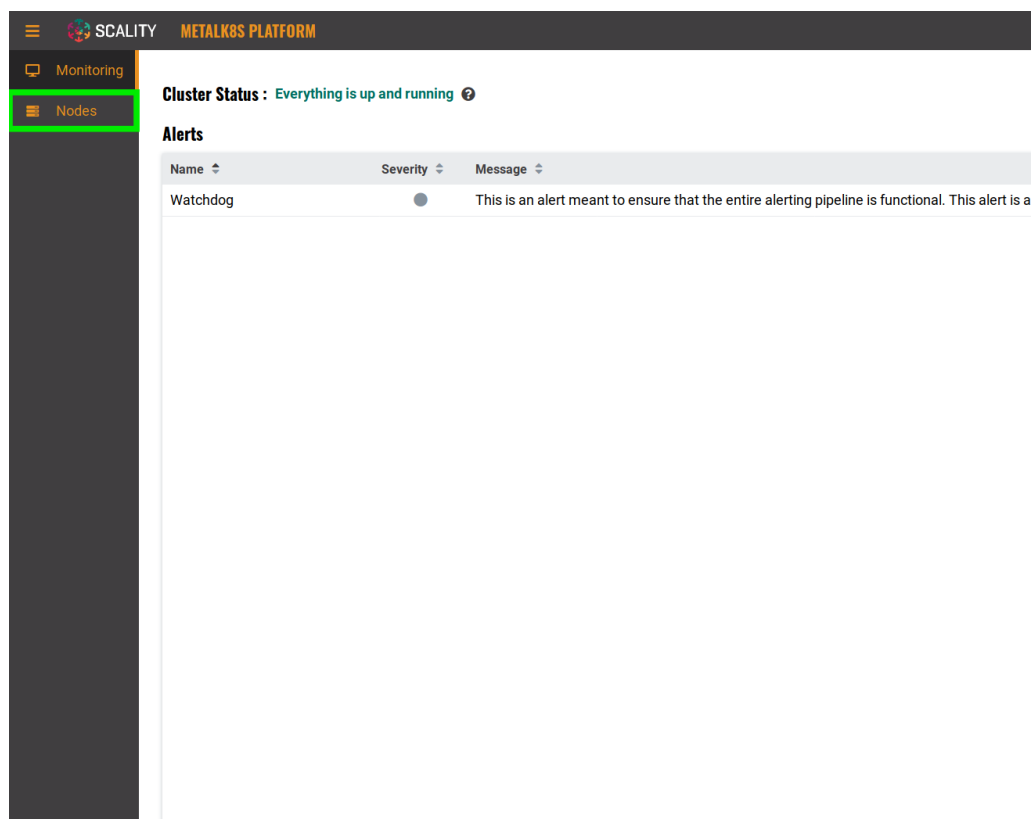
17.3 Volume Management using the UI

This section describes the creation and deletion of MetalK8s **Volume** using the MetalK8s UI. In order to create Volumes you need to have StorageClass objects registered in your cluster. See [StorageClass Creation](#)

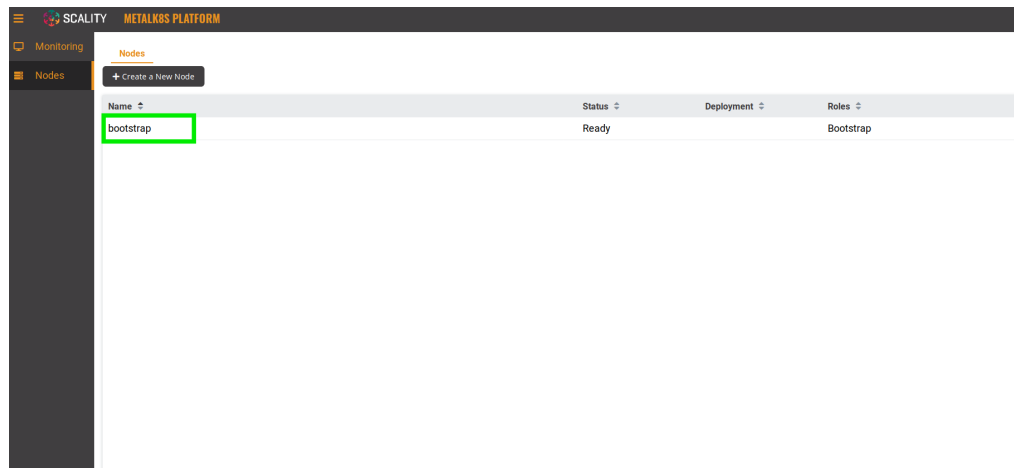
17.3.1 Volume Creation

To access the UI, refer to [this procedure](#)

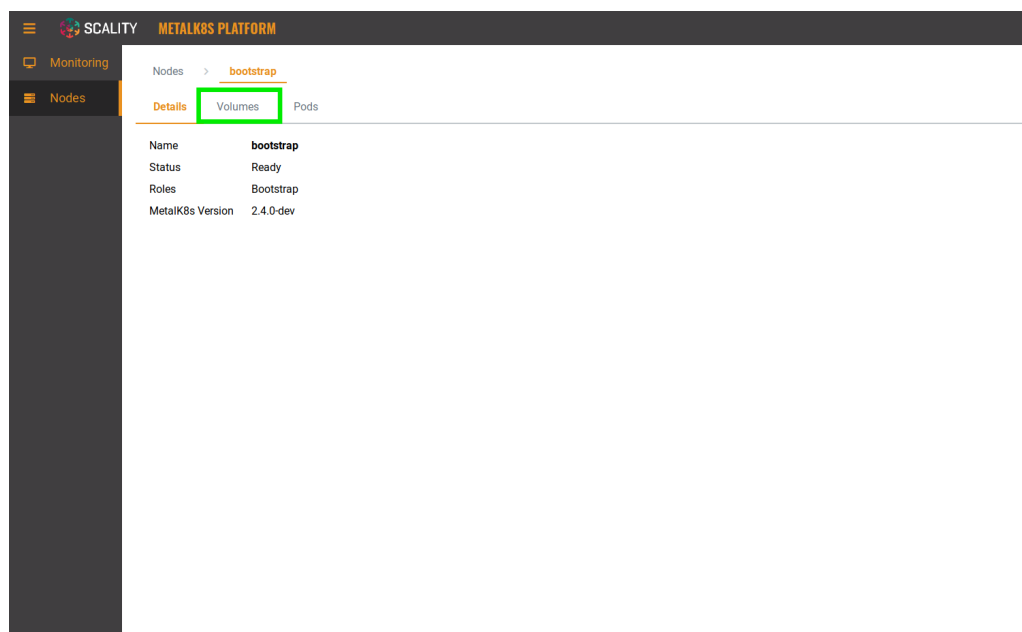
1. Navigate to the **Nodes** list page, by clicking the button in the sidebar:



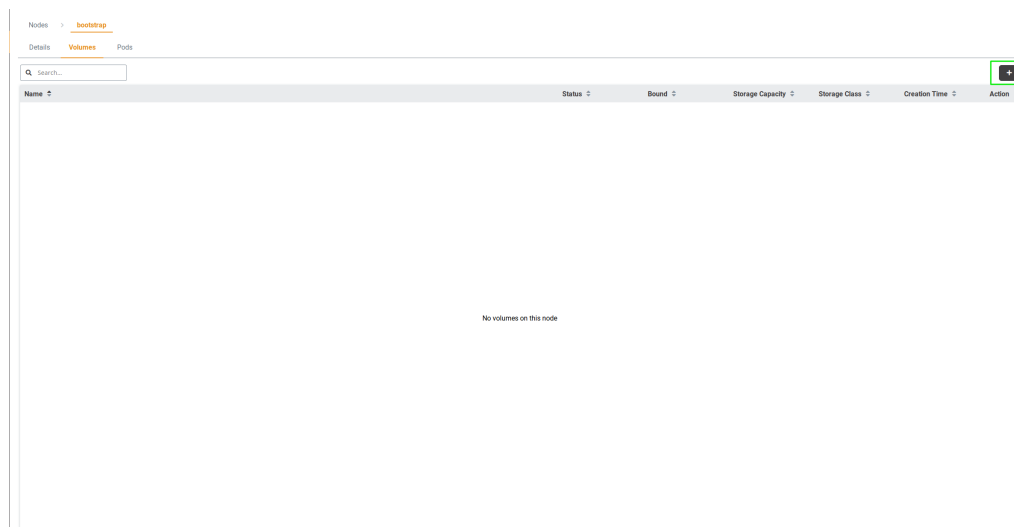
2. From the Node list, select the node you would like to create a volume on



3. Navigate to the **Volumes** tab



4. Click the + button to create a volume



5. Fill out the respective fields

SCALITY METALK8S PLATFORM

Nodes > bootstrap > **Create a New Volume**

Name:

Labels: Add

io.kubernetes/type

Storage Class:

Type:

Device path:

Cancel Create

- **Name:** Denotes the volume name.
- **Labels:** A set of key/value pairs that are used by Persistent Volume Claims to select the right Persistent Volumes.
- **Storage Class:** Refer to the storage class creation page listed here: [StorageClass Creation](#)
- **Type:** MetalK8s currently only supports **RawBlockDevice** and **SparseLoopDevice**.
- **Device path:** Refers to the path of an existing storage device.

6. Finally, click the **Create** button

SCALITY METALK8S PLATFORM

Nodes > bootstrap > **Create a New Volume**

Name:

Labels: Add

io.kubernetes/type

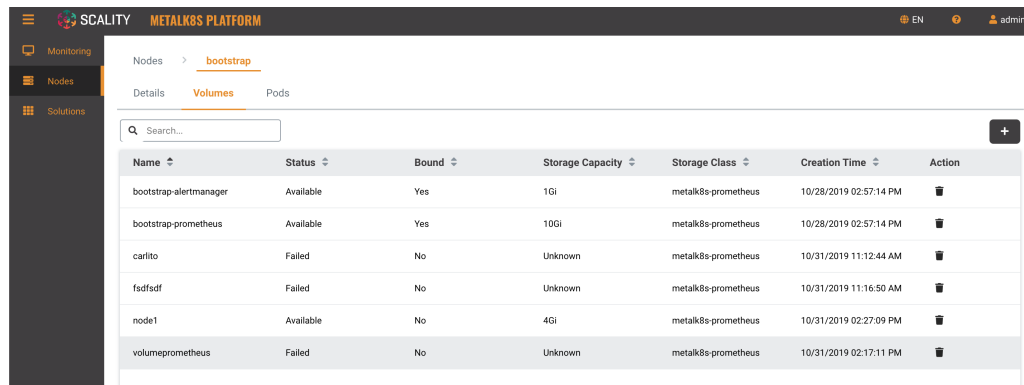
Storage Class:

Type:

Device path:

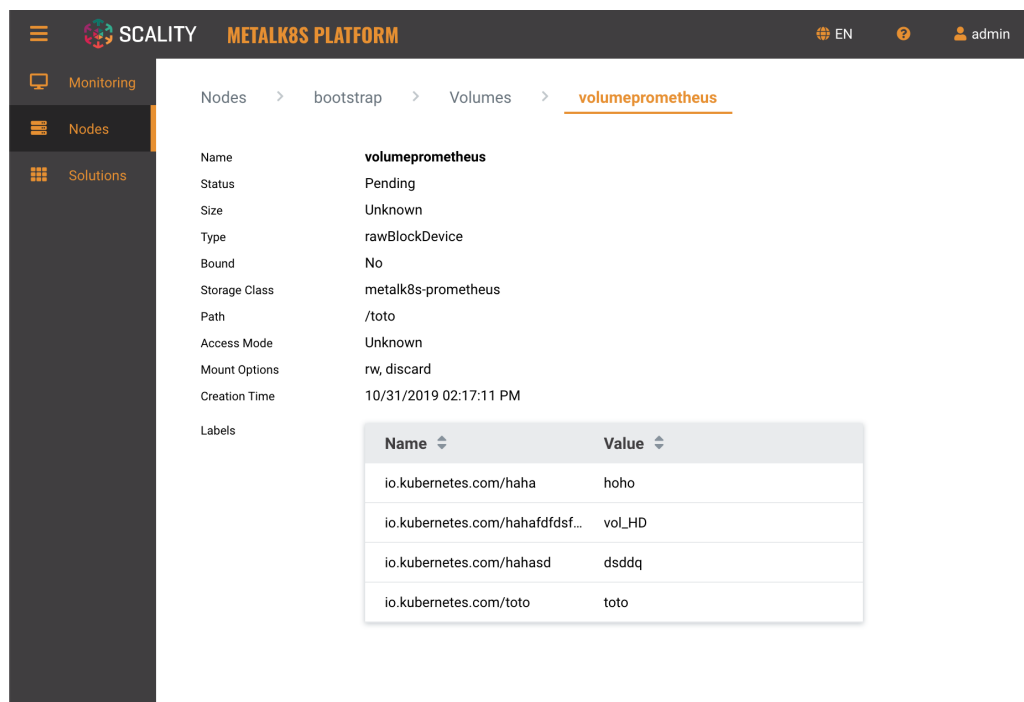
Cancel **Create**

7. You should have a new volume listed in the **Volume list**



Name	Status	Bound	Storage Capacity	Storage Class	Creation Time	Action
bootstrap-alertmanager	Available	Yes	1Gi	metalK8s-prometheus	10/28/2019 02:57:14 PM	
bootstrap-prometheus	Available	Yes	10Gi	metalK8s-prometheus	10/28/2019 02:57:14 PM	
carlito	Failed	No	Unknown	metalK8s-prometheus	10/31/2019 11:12:44 AM	
fsdfsdf	Failed	No	Unknown	metalK8s-prometheus	10/31/2019 11:16:50 AM	
node1	Available	No	4Gi	metalK8s-prometheus	10/31/2019 02:27:09 PM	
volume-prometheus	Failed	No	Unknown	metalK8s-prometheus	10/31/2019 02:17:11 PM	

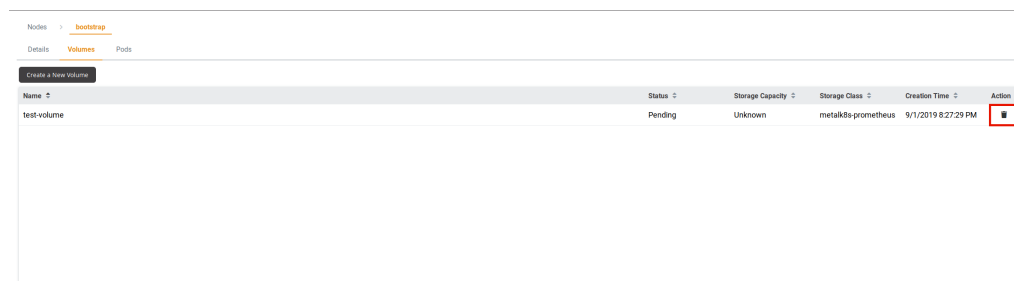
8. If you click on any volume in the Volume list, you will see more information in the Volume detail view:



Name	volume-prometheus										
Status	Pending										
Size	Unknown										
Type	rawBlockDevice										
Bound	No										
Storage Class	metalK8s-prometheus										
Path	/toto										
Access Mode	Unknown										
Mount Options	rw, discard										
Creation Time	10/31/2019 02:17:11 PM										
Labels	<table border="1"> <thead> <tr> <th>Name</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>io.kubernetes.com/haha</td> <td>hoho</td> </tr> <tr> <td>io.kubernetes.com/hahafdfsdf...</td> <td>vol_HD</td> </tr> <tr> <td>io.kubernetes.com/hahasd</td> <td>dsddq</td> </tr> <tr> <td>io.kubernetes.com/toto</td> <td>toto</td> </tr> </tbody> </table>	Name	Value	io.kubernetes.com/haha	hoho	io.kubernetes.com/hahafdfsdf...	vol_HD	io.kubernetes.com/hahasd	dsddq	io.kubernetes.com/toto	toto
Name	Value										
io.kubernetes.com/haha	hoho										
io.kubernetes.com/hahafdfsdf...	vol_HD										
io.kubernetes.com/hahasd	dsddq										
io.kubernetes.com/toto	toto										

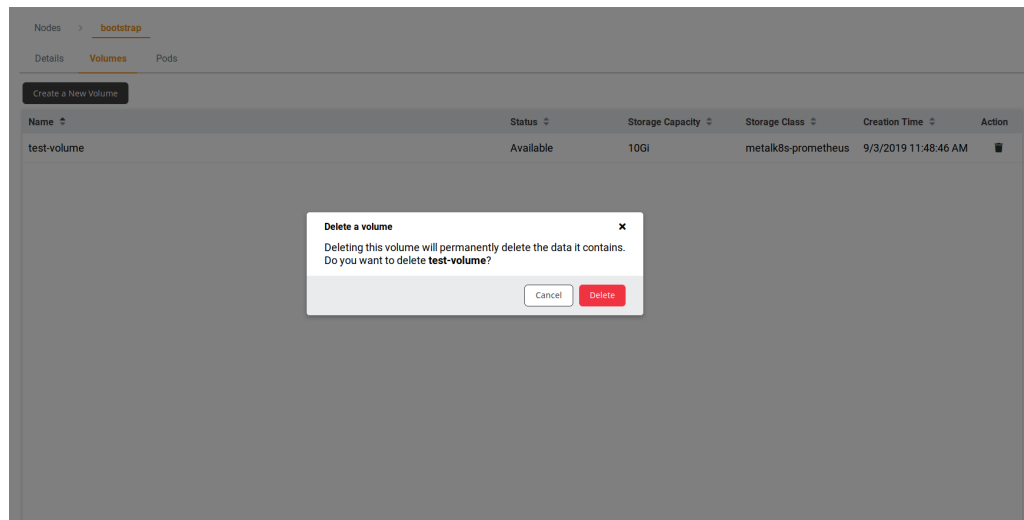
17.3.2 Volume Deletion

1. To delete a volume from the MetalK8s UI, from the volume listing, click the delete button



Name	Status	Storage Capacity	Storage Class	Creation Time	Action
test-volume	Pending	Unknown	metalK8s-prometheus	9/1/2019 8:27:29 PM	

2. Confirm the volume deletion request by clicking the **Delete** button



ACCOUNT ADMINISTRATION

This section highlights **MetalK8s Account Administration** which covers changing the default username and password for some MetalK8s services.

18.1 Administering Grafana

A fresh install of MetalK8s has a Grafana service instance with default credentials: admin / admin. For more information on how to access Grafana, please refer to [this procedure](#)

18.1.1 Changing Grafana username and password

To change the default username and password for Grafana on a MetalK8s cluster, perform the following procedures:

1. Create a file named `patch-secret.yaml` that has the following content:

```
stringData:
  admin-user: <username-in-clear>
  admin-password: <password-in-clear>
```

2. Apply the patch file by running:

```
$ kubectl --kubeconfig /etc/kubernetes/admin.conf patch secrets prometheus-operator-grafana --
↪ patch "$(cat patch-secret.yaml)" -n metalk8s-monitoring
```

3. Now, roll out the new updates for Grafana:

```
$ kubectl --kubeconfig /etc/kubernetes/admin.conf rollout restart deploy prometheus-operator-
↪ grafana -n metalk8s-monitoring
```

4. Access the Grafana instance and authenticate yourself using the new Account credentials.

Warning: During an upgrade or downgrade of a MetalK8s cluster, customized Grafana username and password will be overwritten with default credentials admin / admin.

18.2 Administering MetalK8s GUI, Kubernetes API and Salt API

During installation, MetalK8s configures the Kubernetes API to accept Basic authentication, with default credentials `admin / admin`.

Services exposed by MetalK8s, such as *its GUI* or *Salt API*, rely on the Kubernetes API for authenticating their users. As such, changing the credentials of a Kubernetes API user will also change the credentials required to connect to either one of these services.

18.2.1 Managing Kubernetes API username and password

Warning: The procedures mentioned below must be carried out on every control-plane *Node*, or more specifically, any Node bearing the `node-role.kubernetes.io/master` label.

1. Edit the credentials file located at `/etc/kubernetes/htpasswd`, replacing the username and/or password fields as below:

`<password-in-clear>,<username-in-clear>,123,"system:masters"`

2. Force a restart of the Kubernetes API server:

```
$ crictl stop \  
  $(crictl ps -q --label io.kubernetes.pod.namespace=kube-system \  
    --label io.kubernetes.container.name=kube-apiserver \  
    --state Running)
```

3. Access a service (for example, MetalK8s GUI) and authenticate yourself using the new Account credentials.

Note: Upon changing the username and/or password, a fresh logout then login is required for accessing the MetalK8s GUI.

TROUBLESHOOTING GUIDE

This section highlights some of the common problems users face during and after a MetalK8s installation. If you do not find a solution to a problem you are facing, please reach out to **Scality support** or create a [Github issue](#).

19.1 Bootstrap Installation Errors

19.1.1 Bootstrap Installation fails with no straightforward reason

If during a MetalK8s installation you encounter a failure and the console output does not provide sufficient information in order to pin-point the cause of failure, then re-run the installation with the verbose flag (`--verbose`).

```
root@bootstrap $ /srv/scality/metalk8s-2.4.3/bootstrap.sh --verbose
```

19.1.2 Errors after restarting the Bootstrap node

If you reboot the Bootstrap node and for some reason, some containers (especially the salt-master container) refuses to start then perform the following checks:

- Check and ensure that the **MetalK8s ISO** is mounted properly.

```
[root@bootstrap vagrant]# mount | grep /srv/scality/metalk8s-2.4.3  
/home/centos/metalk8s.iso on /srv/scality/metalk8s-2.4.3 type iso9660 (ro,relatime)
```

- If the ISO is unmounted, run the following command which will check the the status of the ISO file and remount it automatically.

```
[root@bootstrap vagrant]# salt-call state.sls metalk8s.archives.mounted_  
↪ saltenv=metalk8s-2.4.3  
Summary for local  
-----  
Succeeded: 3  
Failed:    0
```

19.1.3 Bootstrap fails and console log is unscrollable

If during a MetalK8s installation, the Bootstrap process fails and the console output is unscrollable then you can consult the Bootstrap logs in `/var/log/metalk8s-bootstrap.log`.

19.2 Account Administration Errors

19.2.1 Forgot the MetalK8s GUI password

If you forgot the MetalK8s GUI username and/or password combination, follow [this procedure](#) to reset or change it.

19.3 General Kubernetes Resource Errors

19.3.1 Pod status shows “CrashLoopBackOff”

If after a MetalK8s installation, you notice some Pods are in a state of “CrashLoopBackOff”, then it means pods are crashing because they start up then immediately exit, thus Kubernetes restarts them and the cycle continues. To get possible clues about this error, run the following commands and inspect the output.

```
[root@bootstrap vagrant]# kubectl -n kube-system describe pods <pod name>
Name:                <pod name>
Namespace:           kube-system
Priority:              2000000000
Priority Class Name:  system-cluster-critical
```

19.3.2 Persistent Volume Claim(PVC) stuck in “Pending” state

If after provisioning a Volume for a Pod (e.g. Prometheus) and the PVC still hangs in a **Pending** state, then try checking the following:

- Check that the volumes have been provisioned and are in a **Ready** state:

```
kubectl describe volume <volume-name>
[root@bootstrap vagrant]# kubectl describe volume test-volume
Name:                <volume-name>
Status:
  Conditions:
    Last Transition Time:  2020-01-14T12:57:56Z
    Last Update Time:     2020-01-14T12:57:56Z
    Status:                True
    Type:                  Ready
```

- Check that a corresponding PersistentVolume exist:

```
[root@bootstrap vagrant]# kubectl get pv
NAME                CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS  STORAGECLASS
↪ AGE              CLAIM
<volume-name>      10Gi      RWO           Retain          Bound  <storage-class-
↪ name>             4d22h     <persistentvolume-claim-name>
```

- Check that the PersistentVolume matches the PersistentVolume Claim constraints (size, labels, storage class) by doing the following:
 - Find the name of your PersistentVolume Claim:


```
[root@bootstrap vagrant]# kubectl get pvc -n <namespace>
NAME                                STATUS  VOLUME          CAPACITY  ACCESS MODES  AGE
↪ STORAGECLASS                     Bound  <volume-name>   10Gi      RWO           ↪
↪ <storage-class-name> 24h
```

- Then check the PersistentVolume Claim constraints if they match:

```
[root@bootstrap vagrant]# kubectl describe pvc <persistentvolume-claim-name> -n <namespace>
Name:          <persistentvolume-claim-name>
Namespace:     <namespace>
StorageClass:  <storage-class-name>
Status:        Bound
Volume:        <volume-name>
Capacity:      10Gi
Access Modes:  RWO
VolumeMode:    Filesystem
```

- If no PersistentVolume exist, then check that the storage operator is up and running.

```
[root@bootstrap vagrant]# kubectl -n kube-system get deployments storage-operator
NAME                READY  UP-TO-DATE  AVAILABLE  AGE
storage-operator   1/1    1           1          4d22h
```

19.3.3 Access to MetalK8s GUI fails with “undefined backend”

If in the cause of using the MetalK8s GUI, you encounter an “undefined backend” error then perform the following checks:

- Check that the Ingress pods are running:

```
[root@bootstrap vagrant]# kubectl -n metalk8s-ingress get daemonsets
NAME                                DESIRED  CURRENT  READY  UP-TO-DATE  AVAILABLE  AGE
↪ NODE SELECTOR                     AGE
nginx-ingress-control-plane-controller 1         1        1      1           1          ↪
↪ node-role.kubernetes.io/master= 4d22h
nginx-ingress-controller             1         1        1      1           1
↪ <none>                             4d22h
```

- Check the Ingress controller logs:

```
[root@bootstrap vagrant]# kubectl logs -n metalk8s-ingress nginx-ingress-control-plane-
↪ controller-ftg6v
-----
NGINX Ingress controller
Release:      0.26.1
Build:        git-2de5a893a
Repository:   https://github.com/kubernetes/ingress-nginx
nginx version: openresty/1.15.8.2
```

19.3.4 Pod and Service CIDR conflicts

If after installation of a MetalK8s cluster you notice that Pod-to-Pod communication has routing problems, perform the following:

- Check the configured values for the internal Pod and Service networks:

```
[root@bootstrap vagrant]# salt-call pillar.get networks
local:
-----
control_plane:
  172.21.254.0/28
pod:
  10.233.0.0/16
service:
  10.96.0.0/12
workload_plane:
  172.21.254.32/27
```

Make sure the configured IP ranges (CIDR notation) do not conflict with your infrastructure.

Todo:

- Add Salt master/minion logs, and explain how to run a specific state from the Salt master.
 - Add troubleshooting for networking issues.
-

Part III

Developer Guide

ARCHITECTURE DOCUMENTS

20.1 Authentication

20.1.1 Context

Currently, when we deploy MetalK8s we pre-provision a super admin user with a username/password pair. This implies that anyone wanting to use the K8S/Salt APIs needs to authenticate using this single super admin user.

Another way to access the APIs is by using the K8S admin certificate which is stored in `/etc/kubernetes/admin.conf`. We could also manually provision other users, their corresponding credentials as well as role bindings but this current approach is inflexible to operate in production setups and security is not guaranteed since username/password pairs are stored in cleartext.

We would atleast like to be able to add different users with different credentials and ideally integrate K8S authentication system with external an identity provider.

Managing K8S role binding between user/groups High level roles and K8S roles is not part of this specification.

20.1.2 Requirements

Basically, we are talking about:

- Being able to provision users with an local Identity Provider (IDP)
- Being able to integrate with an external IDP

Integration with LDAP and Microsoft Active Directory(AD) are the most important ones to support.

20.1.3 User Stories

Pre-provisioned user and password change

In order to stay aligned with many other applications, it would make sense to have a pre-provisioned user with all privileges (kind of super admin) and pre-provisioned password so that it is easy to start interacting with the system through various admin UIs. Whatever UI this user opens for the first time, the system should ask him/her to change the password for obvious security reasons.

User Management with local IdP

As an IT Generalist, I want to provision/edit users and high-level roles. The MetalK8s high-level roles are:

- Cluster Admin role
- Solution Admin role
- Read Only

This is done from CLI with well-documented procedure. Entered passwords are never visible and encrypted when stored in local IDP DB. The CLI tool enables to add/delete and edit passwords and roles.

External IDP Integration

As an IT Generalist, I want to leverage my organisation's IDP to reuse already provisioned users & groups. The way we do that integration is through a CLI tool which does not require to have deep knowledge in K8S or in any local IDP specifics. When External IDP Integration is set up, we can always use local IDP to authenticate.

Authentication check

UI should make sure the user is well authenticated and if not, redirect to the local IDP login page. In the local IDP login page, the user should choose between authenticating with local IDP or with external IDP. If no external IDP is configured, no choice is presented to the user. This local IDP login page should be styled so that it looks like any other MetalK8s or solutions web pages. All admin UIs should share the same IDP.

Configuration persistence

Upgrading or redeploying MetalK8s should not affect configuration that was done earlier (i.e. local users and credentials as well as external IDP integration and configuration)

SSO between Admin UIs

Once IDP is in place and users are provisioned, one authenticated user can easily navigate to the other admin UIs without having to re-authenticate.

20.1.4 Open questions

- Authentication across multiple sites
- SSO across MetalK8s and solutions Admin UIs and other workload Management UIs
- Our customers may want to collect some statistics out of our Prometheus instances. This API could be authenticated using OIDC, using an OIDC proxy, or stay unauthenticated. One should consider the following factors:
 - the low sensitivity of the exposed data
 - the fact that it is only exposed on the control-plane network
 - the fact that most consumers of Prometheus stats are not human (e.g. Grafana, a federating Prometheus, scripts and others), hence not well-suited for performing the OIDC flow

20.1.5 Design Choices

Dex is chosen as an Identity Provider(IdP) in MetalK8s based on the above [Requirements](#) for the following reasons:

- Dex's support for multiple plugins enable integrating the OIDC flow with existing user management systems such as Active Directory, LDAP, SAML and others.
- Dex can be seamlessly deployed in a Kubernetes cluster.
- Dex provides access to a highly customizable UI which is a step closer to good user experience which we advocate for.
- Dex can act as a fallback Identity Provider in cases where the external providers become unavailable or are not configured.

Rejected design choices

Static password file Vs OpenID Connect

Using static password files involves adding new users by updating a static file located on every control-plane Node. This method requires restarting the Kubernetes API server for every new change introduced.

This was rejected since it is inflexible to operate, requires storing user credentials and there is no support for a pluggable external identity provider such as LDAP.

X.509 certificates Vs OpenID Connect

Here, each user owns a signed certificate that is validated by the Kubernetes API server.

This approach is not user-friendly that is each certificate has to be manually signed. Providing certificates for accessing the MetalK8s UI needs much more efforts since these certificates are browser incompatible. Using certificates is tedious since the certificate revocation process is also cumbersome.

Keycloak Vs Dex

Both systems use OpenID Connect(OIDC) to authenticate a user using a standard OAuth2 flow.

They both offer the ability to have short lived sessions so that user access can be rotated with minimum efforts.

Finally, they both provide a means for identity management to be handled by an external service such as LDAP, Active Directory, SAML and others.

Why not Keycloak?

Keycloak while offering similar features as Dex and even much more was rejected for the following reasons:

- Keycloak is complex to operate(requires its own standalone database) and manage(frequent database backups are required).
- Currently, no use case exist for implementing a sophisticated Identity Provider like Keycloak when the minimal Identity Provider from Dex is sufficient.

Note that, Keycloak is considered a future fallback Identity Provider if the need ever arises from a customer standpoint.

Unexploited choices

- [Guard](#)

A Kubernetes webhook authentication server by AppsCode, allowing you to log into your Kubernetes cluster by using various identity providers such as LDAP.

- [ORY Hydra](#)

It's an OpenID Connect provider optimized for low resource consumption. ORY Hydra is not an identity provider but it is able to connect to existing identity providers.

20.1.6 Implementation Details

Iteration 1

- Using Salt, generate self-signed certificates needed for Dex deployment
- Deploy Dex in MetalK8s from the official **Dex Charts** while making use of the generated certificates above
- Provision an admin superuser
- Configure Kubernetes API server flags to use Dex
- Expose Dex on the control-plane using Ingress
- Print the admin super user credentials to the CLI after MetalK8s bootstrap is complete
- Implement MetalK8s UI integration with Dex
- Theme the Dex GUI to match MetalK8s UI specs(optional for iteration 1)

Iteration 2

- Provide documentation on how to integrate with these external Identity Providers especially LDAP and Microsoft Active Directory.

Iteration 3

- Provide Single sign-on(SSO) for Grafana
- Provide SSO between admin UIs

Iteration 4

- Provide a CLI command to change the default superuser password as a prompt after bootstrap
- Provide a CLI for user management and provisioning

The following operations will be supported using the CLI tool:

- Create users password
- List existing passwords
- Delete users password
- Edit existing password

The CLI tool will also be used to create MetalK8s dedicated roles as already specified in the requirements section of this document. (see high-level roles from the requirements document)

Since it is not advisable to perform the above mentioned operations at the Dex ConfigMap level, using the Dex gRPC API could be the way to go.

Iteration 5

- Demand for a superuser's default password change upon first UI access
- Provide UI integration that performs similar CLI operations for user management and provisioning

This means from the MetalK8s UI, a Cluster administrator should be able to do the following:

- Create passwords for users
- List existing passwords
- Delete users password
- Edit existing password

Note: This iteration is completely optional for reasons being that the Identity Provider from Dex acts as a fallback for Kubernetes Administrators who do not want to use an external Identity Provider(mostly because they have a very small user store).

20.1.7 Documentation

In the Operational Guide:

- Document the predefined dex roles(Cluster Admin role, Solution Admin role, Read Only role), their access levels and how to create them.
- Document how to create users and the secrets associated to them.
- Document how to integrate Dex with external Identity Providers such as LDAP and Microsoft Active Directory.

In the Installation/Quickstart Guide

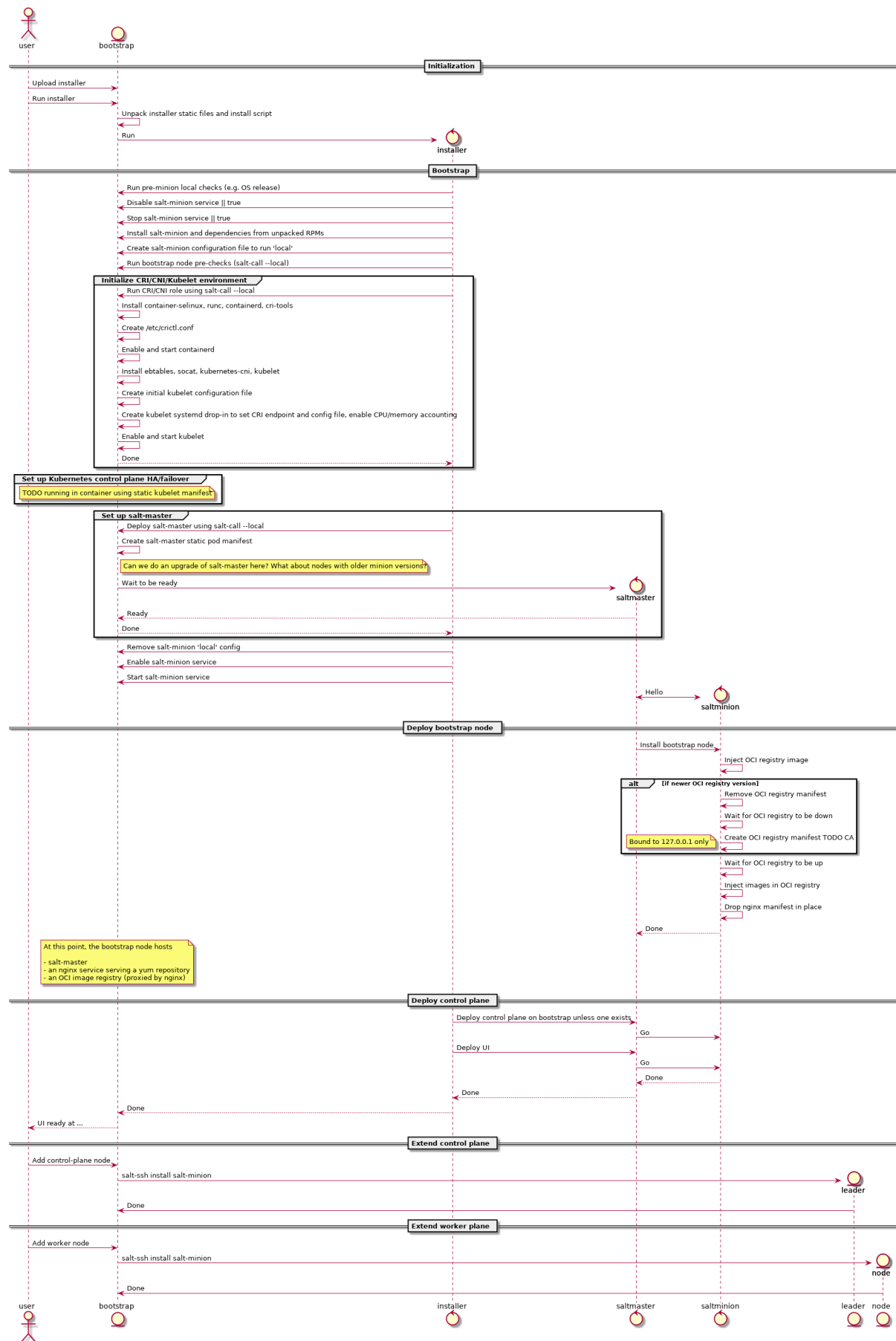
- Document how to setup/change the superuser password

20.1.8 Test Plan

We could add some automated end-to-end tests for Dex user creation, and deletion using the CLI and then setup a mini-lab on scalability cloud to try out the UI integration.

20.2 Deployment

Here is a diagram representing how MetalK8s orchestrates deployment on a set of machines:



20.2.1 Some notes

- The intent is for this installer to deploy a system which looks exactly like one deployed using kubeadm, i.e. using the same (or at least highly similar) static manifests, cluster ConfigMaps, RBAC roles and bindings, ...

The rationale: at some point in time, once kubeadm gets easier to embed in larger deployment mechanisms, we want to be able to switch over without too much hassle.

Also, kubeadm applies best-practices so why not follow them anyway.

Configuration

To launch the bootstrap process, some input from the end-user is required, which can vary from one installation to another:

- CIDR (i.e. `x.y.z.w/n`) of the control plane networks to use
Given these CIDR, we can find the address on which to bind services like etcd, kube-apiserver, kubelet, salt-master and others.
These should be existing networks in the infrastructure to which all hosts are connected.
This is a list of CIDRs, which will be tried one after another, to find a matching local interface (i.e. hosts comprising the cluster may reside in different subnets, e.g. control plane in VMs, workload plane on physical infrastructure).
- CIDRs (i.e. `x.y.z.w/n`) of the workload plane networks to use
Given these CIDRs, we can find the address to be used by the CNI overlay network (i.e. Calico) for inter-Pod routing.
This can be the same as the control plane network.
- CIDR (i.e. `x.y.z.w/n`) of the Pod overlay network
Used to configure the Calico IPPool. This must be a non-existing network in the infrastructure.
Default: `10.233.0.0/16`
- CIDR (i.e. `x.y.z.w/n`) of the Service network
Default: `10.96.0.0/12`

Firewall

We assume a host-based firewall is used, based on `firewalld`. As such, for any service we deploy which must be accessible from the outside, we must set up an appropriate rule.

We assume SSH access is not blocked by the host-based firewall.

These services include:

- HTTPS on the bootstrap node, for nginx fronting the OCI registry and serving the yum repository
- salt-master on the bootstrap node
- etcd on control-plane / etcd nodes
- kube-apiserver on control-plane nodes
- kubelet on all cluster nodes

20.3 Monitoring

This document describes the monitoring features included in MetalK8s.

Todo: Describe the monitoring stack ([#1075](#)), include quick explanation in quickstart guide.

20.4 Alerting Functionalities

20.4.1 Context

MetalK8s is automatically deploying Prometheus, Alertmanager and a set of predefined alert rules. In order to leverage Prometheus and Alertmanager functionalities, we need to explain, in the documentation, how to use it. In a later stage, those functionalities will be exposed through various administration and alerting UIs, but for now, we want to provide our administrator with enough information in order to use very basic alerting functionalities.

Requirements

As a MetalK8s administrator, I want to list or know the list of alert rules that are deployed on MetalK8s Prometheus cluster, In order to identify on what specific rule I want to be alerted.

As a MetalK8s administrator, I want to set notification routing and receiver for a specific alert, In order to get notified when such alert is fired The important routing to support are email, slack and pagerduty.

As a MetalK8s administrator, I want to update thresholds for a specific alert rule, In order to adapt the alert rule to the specificities and performances of my platform.

As a MetalK8s administrator, I want to add a new alert rule, In order to monitor a specific KPI which is not monitored out of the box by MetalK8s.

As a MetalK8s administrator, I want to inhibit an alert rule, In order to skip alerts in which I am not interested.

As a MetalK8s administrator, I want to silence an alert rule for a certain amount of time, In order to skip alert notifications during a planned maintenance operation.

Warning: In all cases, when MetalK8s administrator is upgrading the cluster, all listed customizations should remain.

Note: Alertmanager configuration documentation is available [here](#)

20.5 Requirements

20.5.1 Deployment

Mimick Kubeadm

A deployment based on this solution must be as close to a *kubeadm*-managed deployment as possible (though with some changes, e.g. non-root services). This should, over time, allow to actually integrate *kubeadm* and its ‘business-logic’ in the solution.

Fully Offline

It should be possible to install the solution in a fully offline environment, starting from a set of ‘packages’ (format to be defined), which can be brought into the environment using e.g. a DVD image. It must be possible to validate the provenance and integrity of such image.

Fully Idempotent

After deployment of a specific version of the solution in a specific configuration / environment, it shall be possible to re-run this deployment, which should cause no changes to the system(s) involved.

Single-Server

It must be possible to deploy the solution on a single server (without any expectations w.r.t. availability, of course).

Scale-Up from Single-Server Deployment

Given a single-server deployment, it must be possible to scale up to multiple nodes, including control plane as well as workload plane.

Installation == Upgrade

There shall be no difference between ‘installation’ of the solution vs. upgrading a deployment, from a logical point of view. Of course, where required, particular steps in the implementation may cause other actions to be performed, or specific steps to be skipped.

Rolling Upgrade

When upgrading an environment, this shall happen in ‘rolling’ fashion, always cordoning, draining, upgrading and uncordoning nodes.

Handle CentOS Kernel Memory Accounting

The solution must provide versions of *runc* and *kubelet* which are built to include the fixes for the *kmem* leak issues found on CentOS/RHEL systems.

See:

- <https://github.com/kubernetes/kubernetes/issues/61937>
- <https://github.com/kubernetes/kubernetes/pull/72114#issuecomment-454953077>
- <https://github.com/kubernetes/kubernetes/pull/72998#issuecomment-455512443>

At-Rest Encryption

Data stored by Kubernetes must be encrypted at-rest (TBD which kind of objects).

Node Labels

Nodes in the cluster can be properly labeled, e.g. including availability zone information.

Vagrant

For evaluation purposes, it should be possible to set up a cluster in a *Vagrant* environment, in a fully automated fashion.

20.5.2 Runtime

No Root

All services, including those managed by *kubelet*, must run as a non-root user, if possible. This user must be provisioned as a system user/group. E.g., for the *etcd* service, despite being managed by *kubelet* using a static Pod manifest, a suitable *etcd* user and group should be created on the system, */var/lib/etcd* (or similar) must be owned by this user/group, and the Pod manifest shall specify the *etcd* process must run as said UID/GID.

SELinux

The solution may not require SELinux to be disabled or put in permissive mode.

It must, however, be possible to configure workload-plane nodes to be put in SELinux disabled or permissive mode, if applications running in the cluster can't support SELinux.

Read-Only Containers

All containers as deployed by the solution must be fully immutable, i.e. read-only, with *EmptyDir* volumes as temporary directories where required.

Environment

The solution must support CentOS 7.6.

CRI

The solution shall not depend on Docker to be available on the systems, and instead rely on either *containerd* or *cri-o*. TBD which one.

OIDC

For ‘human’ authentication, the solution must integrate with external systems like Active Directory. This may be achieved using OIDC.

For environments in which an external directory service is not available, static users can be configured.

20.5.3 Distribution

No Random Binaries

Any binary installed on a host system must be installed by a system package (e.g. RPM) through the system package manager (e.g. yum).

Tagged Generated Files

Any file generated during deployment (e.g. configuration files) which are not required to be part of a system package (i.e. they are installation-specific) should, if possible, contain a line (as a comment, a preamble, ...) describing the file was generated by this project, including project version (TBD, given idempotency) and timestamp (TBD, given idempotency).

Container Images

All container (OCI) images must be built from a well-known base image (e.g. upstream CentOS images), which shall be based on a digest and parametrized during build (which allows for easy upgrades of all images when required).

During build, only ‘system’ packages (e.g. RPM) can be installed in the container, using the system package manager (e.g. CentOS), to ensure the ability to validate provenance and integrity of all files part of said image.

All containers should be properly labeled (TODO), and define suitable *PORT* and *ENTRYPOINT* directives.

20.5.4 Networking

Zero-Trust Networking: Transport

All over-the-wire communication must be encrypted using TLS.

Zero-Trust Networking: Identity

All over-the-wire communication must be validated by checking server identity and, where sensible, validating client/peer identity.

Zero-Trust Networking: Certificate Scope

Certificates for different ‘realms’ must come from different CA chains, and can’t be shared across multiple hosts.

Zero-Trust Networking: Certificate TTL

All issued certificates must have a reasonably short time-to-live and, where required, be automatically rotated.

Zero-Trust Networking: Offline Root CAs

All root CAs must be kept offline, or be password-protected. For automatic certificate creation, intermediate CAs (online, short/medium-lived, without password protection) can be used. These need to be rotated on a regular basis.

Zero-Trust Networking: Host Firewall

The solution shall deploy a host firewall (e.g., using *firewalld*) and configure it accordingly (i.e., open service ports where applicable).

Furthermore, if possible, access to services including *etcd* and *kubelet* should be limited, e.g. to *etcd* peers or control-plane nodes in the case of *kubelet*.

Zero-Trust Networking: No Insecure Ports

Several Kubernetes services can be configured to expose an unauthenticated endpoint (sometimes for read-only purposes only). These should always be disabled.

Zero-Trust Networking: Overlay VPN (Optional)

Encryption and mutual identity validation across nodes for the CNI overlay, bringing over-the-wire encryption for workloads running inside Kubernetes without requiring a service mesh or per-application TLS or similar, if required.

DNS

Network addressing must, primarily, be based on DNS instead of IP addresses. As such, certificate SANs should not contain IP addresses.

Server Address Changes

When a server receives a different IP address after a reboot (but can still be discovered through an updated DNS entry), it must be possible to reconfigure the deployment accordingly, with as little impact as possible (i.e., requiring as little changes as possible). This related to the *DNS* section above.

For some services, e.g. *keepalived* configuration, IP addresses are mandatory, so these are permitted.

Multi-Homed Servers

A deployment can specify subnet CIDRs for various purposes, e.g. control-plane, workload-plane, etcd, ... A service part of a specific 'plane' must be bound to an address in said 'plane' only.

Availability of kube-apiserver

kube-apiserver must be highly-available, potentially using failover, and (optionally) made load-balanced. I.e., in a deployment we either run a service like *keepalived* (with VRRP and a VIP for HA, and IPVS for LB), or there's a site-local HA/LB solution available which can be configured out-of-band.

E.g. for *kube-apiserver*, its */healthz* endpoint can be used to validate liveness and readiness.

Provide LoadBalancer Services

The solution brings an optional controller for *LoadBalancer* services, e.g. MetalLB. This can be used to e.g. front the built-in *Ingress* controller.

In environments where an external load-balancer is available, this can be omitted and the external load-balancer can be integrated in the Kubernetes infrastructure (if supported), or configured out-of-band.

Network Configuration: MTU

Care shall be taken to set networking configuration, e.g. MTU sizes, properly across the cluster and the services relying on it (e.g. the CNI).

Network Configuration: IPIP

Unless required, 'plain' networking must be used instead of tunnels, i.e., when using Calico, IPIP should only be used in cross-subnet networking.

Network Configuration: BGP

In environments where routing configuration using BGP can be achieved, this should be feasible for MetalLB-managed services, as well as Calico routing, in turn removing the need for IPIP usage.

IPv6

TODO

20.5.5 Storage

TODO

20.5.6 Batteries-Included

Similar to MetalK8s 1.x, the solution comes 'batteries included'. Some aspects of this, including optional HA/LB for *kube-apiserver* and *LoadBalancer* Services using MetalLB have been discussed before.

Metrics and Alerting: Prometheus

The solution comes with *prometheus-operator*, including *ServiceMonitor* objects for provisioned services, using exporters where required.

Node Monitoring: node_exporter

The solution comes with *node_exporter* running on the hosts (or a *DaemonSet*, if the volume usage restriction can be fixed).

Node Monitoring: Platform

The solution integrates with specific platforms, e.g. it deploys an HPE iLO exporter to capture these metrics.

Node Monitoring: Dashboards

Dashboards for collected metrics must be deployed, ideally using some *grafana-operator* for extensibility sake.

Logging

The solution comes with log aggregation services, e.g. *fluent-bit* and *fluentd*. Either a storage system for said logs is deployed as part of the cluster (e.g. Elasticsearch with Kibana, Curator, Cerebro), or the aggregation system is configured to ingest into an environment-specific aggregation solution, e.g. Splunk.

Container Registry

To support fully-offline environments, this is required.

System Package Repository

See above.

Tracing Infrastructure (Optional)

The solution can deploy an OpenTracing-compatible aggregation and inspection service.

Backups

The solution ensures backups of core data (e.g. *etcd*) are made, at regular intervals as well as before a cluster upgrade. These can be stored on the cluster node(s), or on a remote storage system (e.g. NFS volume).

20.6 Solutions

20.6.1 Context

As for now, if we want to deploy applications on a MetalK8s cluster, it's achievable by applying manifest through `kubectl apply -f manifest.yaml` or using [Helm](#) with charts.

These approaches work, but for an offline environment, the user must first inject all the needed images in [containerd](#) on every nodes. Plus, this requires some Kubernetes knowledge to be able to install an application.

Moreover, there is no control on what's deployed, so it is difficult to enforce certain practices or provide tooling to ease deployment or lifecycle management of these applications.

We also want MetalK8s to be responsible for deploying applications to keep Kubernetes as an implementation detail for the end user and as so the user does not need any specific knowledge around it to manage its applications.

20.6.2 Requirements

- Ability to orchestrate the deployment and lifecycle of complex applications.
- Support offline deployment, upgrade and downgrade of applications with arbitrary container images.
- Applications must keep running after a node reboot or a rescheduling of the containers.
- Check archives integrity, validity and authenticity.
- Handle multiple instance of an application with same or different versions.
- Enforce practices (Operator pattern).
- Guidelines for applications developers.

20.6.3 User Stories

Application import

As a cluster administrator, I want to be able to import an application archive using a CLI tool, to make the application available for deployment.

Application deployment and lifecycle

As an application administrator, I want to manage the deployment and lifecycle (upgrade/downgrade/scaling/configuration/deletion) of an application using either a UI or through simple CLI commands (both should be available).

Multiple instances of an application

As an application administrator, I want to be able to deploy both a test and a prod environments for an application, without collision between them, so that I can qualify/test the application on the test environment.

Application development

As a developer, I want to have guidelines to follow to develop an application.

Application packaging

As a developer, I want to have documentation to know how to package an application.

Application validation

As a developer, I want to be able to know that a packaged application follows the requirements and is valid using a CLI tool.

20.6.4 Design Choices

Solutions

What's a Solution

It's a packaged Kubernetes application, archived as an ISO disk image, containing:

- A set of OCI images to inject in MetalK8s image registry
- An Operator to deploy on the cluster
- A UI to manage and monitor the application (optional)

Solution Configuration

MetalK8s already uses a BootstrapConfiguration object, stored in `/etc/metalk8s/bootstrap.yaml`, to define how the cluster should be configured from the bootstrap node, and what versions of MetalK8s are available to the cluster.

In the same way, we will use a SolutionsConfiguration object, stored in `/etc/metalk8s/solutions.yaml`, to declare which Solutions are available to the cluster, from the bootstrap node.

Here is how it will look:

```
apiVersion: solutions.metalk8s.scality.com/v1alpha1
kind: SolutionsConfiguration
archives:
  - /path/to/solution/archive.iso
active:
  solution-name: X.Y.Z-suffix (or 'latest')
```

In this configuration file, no explicit information about the contents of archives should appear. When read by Salt at import time, the archive metadata will be discovered from the archive itself using a well-known file (`/product.txt`), which needs to define at least these 2 lines:

```
NAME=<solution-name>
VERSION=<solution-version>
```

It will also be possible to define specific configuration for a Solution, using a `config.yaml` file at the root of the archive, with the following format:

```
apiVersion: solutions.metalk8s.scality.com/v1alpha1
kind: SolutionConfig
operator:
  image:
    name: solution-name-operator
    tag: 1.0.0
ui:
  image:
    name: solution-name-ui
    tag: 1.0.0
customApiGroups:
  - custom-api-group
```

Todo: Operator roles management will be handled differently in the future, see #2389 for details.

Todo: `config.yaml` and `product.txt` should be merged in only one file such as a `manifest.yaml`, see #2422 for details.

These configuration files will be read by a Salt external pillar module, which will permit the consumption of them by Salt modules and states.

The external pillar will be structured as follows:

```
metalk8s:
  solutions:
    available:
      solution-name:
        - active: True
          archive: /path/to/solution/archive.iso
          config:
            # The content of config.yaml (or a default configuration
            # computed from product.txt, if there is no such file).
            customApiGroups:
              - custom-api-group
            operator:
              image:
                name: solution-name-operator
                tag: 1.0.0
            ui:
              image:
                name: solution-name-ui
                tag: 1.0.0
            id: solution-name-1.0.0
            mountpoint: /srv/scality/solution-name-1.0.0
            name: Solution Name
            version: 1.0.0
          config:
            # Content of /etc/metalk8s/solutions.yaml (SolutionsConfiguration)
            apiVersion: solutions.metalk8s.scality.com/v1alpha1
            kind: SolutionsConfiguration
            archives:
              - /path/to/solutions/archive.iso
            active:
              solution-name: X.Y.Z-suffix (or 'latest')
            environments:
              # Fetched from namespaces with label
```

(continues on next page)

(continued from previous page)

```
# solutions.metalk8s.scality.com/environment
env-name:
# Fetched from namespace annotations
# solutions.metalk8s.scality.com/environment-description
description: Environment description
namespaces:
  solution-a-namespace:
    # Data of metalk8s-environment ConfigMap from this namespace
    config:
      solution-name: 1.0.0
  solution-b-namespace:
    config: {}
```

Archive format

The archive will be packaged as an ISO image.

We chose the ISO image format instead of a compressed archive, like a tarball, because we wanted something easier to inspect without having to uncompress it.

It could also be useful to be able to burn it on a CD, when being in an offline environment and/or with strong security measures (read-only device that can be easily verified).

Solution archive will be structured as follows:

```
.
├── config.yaml
├── images
│   ├── some_image_name
│   │   ├── 1.0.1
│   │   │   ├── <layer_digest>
│   │   │   ├── manifest.json
│   │   │   └── version
│   └── registry-config.inc
├── operator
│   ├── deploy
│   │   └── crds
│   │       └── some_crd_name.yaml
└── product.txt
```

OCI Images registry

Every container images from Solution archive will be exposed as a single repository through MetalK8s registry. The name of this repository will be computed from the product information <NAME>-<VERSION>.

Operator Configuration

Each Solution Operator needs to implement a `--config` flag which will be used to provide a yaml configuration file. This configuration will contain the list of available images for a Solution and where to fetch them. This configuration will be formatted as follows:

```
apiVersion: solutions.metalk8s.scality.com/v1alpha1
kind: OperatorConfig
repositories:
  <solution-version-x>:
    - endpoint: metalk8s-registry/<solution-name>-<solution-version-x>
      images:
        - <image-x>:<tag-x>
```

(continues on next page)

(continued from previous page)

```

    - <image-y>:<tag-y>
  <solution-version-y>:
    - endpoint: metalk8s-registry/<solution-name>-<solution-version-y>
      images:
        - <image-x>:<tag-x>
        - <image-y>:<tag-y>

```

Solution environment

Solutions will be deployed into an Environment, which is basically a namespace or a group of namespaces with a specific label `solutions.metalk8s.scality.com/environment`, containing the Environment name, and an annotation `solutions.metalk8s.scality.com/environment-description`, providing a human readable description of it:

```

apiVersion: v1
kind: Namespace
metadata:
  annotations:
    solutions.metalk8s.scality.com/environment-description: <env-description>
  labels:
    solutions.metalk8s.scality.com/environment: <env-name>
  name: <namespace-name>

```

It allows to run multiple instances of a Solution, optionally with different versions, on the same cluster, without collision between them.

Each namespace in an environment will have a [ConfigMap](#) `metalk8s-environment` deployed which will describe what an environment is composed of (Solutions and versions). This [ConfigMap](#) will then be consumed by Salt to deploy Solutions Operators and UIs.

This [ConfigMap](#) will be structured as follows:

```

apiVersion: solutions.metalk8s.scality.com/v1alpha1
kind: ConfigMap
metadata:
  name: metalk8s-environment
  namespace: <namespace-name>
data:
  <solution-x-name>: <solution-x-version>
  <solution-y-name>: <solution-y-version>

```

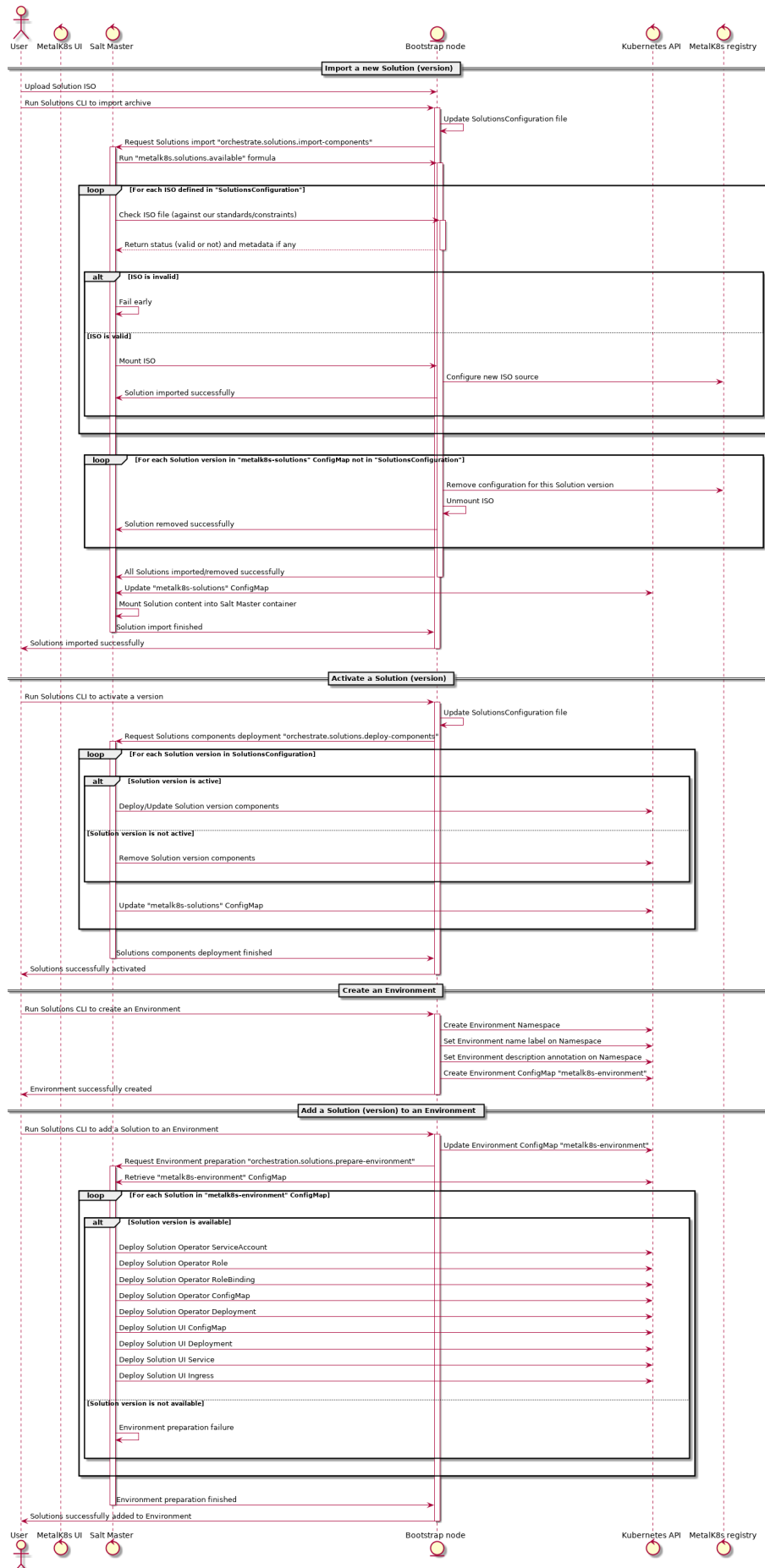
Environments will be created by a CLI tool or through the Solution UI (both should be available), prior to deploy Solutions.

Solution management

We will provide CLI and UI to import, deploy and handle the whole lifecycle of a Solution. These tools are wrapper around Salt formulas.

Interaction diagram

We include a detailed interaction sequence diagram for describing how MetalK8s will handle user input when deploying / upgrading Solutions.



20.6.5 Rejected design choices

CNAB

The Cloud Native Application Bundle (CNAB) is a standard packaging format for multi-component distributed applications. It basically offers what MetalK8s Solution does, but with the need of an extra container with almost full access to the Kubernetes cluster and that's the reason why we did choose to not use it.

We also want to enforce some practices (Operator, UI, ...) in Solutions and this is not easily doable using it.

Moreover, CNAB is a pretty young project and has not yet been adopted by a lot of people, so it's hard to predict its future.

20.6.6 Implementation Details

Iteration 1

- Solution example, this is a fake application, with no other goal than allowing testing of MetalK8s Solutions tooling.
- Salt formulas to manage Solution (deployment and lifecycle).
- Tooling around Salt formulas to ease Solutions management (simple shell script).
- MetalK8s UI to manage Solution.
- Solution automated tests (deployment, upgrade/downgrade, deletion, ...) in post-merge.

Iteration 2

- MetalK8s CLI to manage Solutions (supersedes shell script & wraps Salt call).
- Integration into monitoring tools (Grafana dashboards, Alerting, ...).
- Integration with the identity provider (Dex).
- Tooling to validate integrity & validity of Solution ISO (checksum, layout, valid manifests, ...).
- Multiple CRD versions support (see #2372).

20.6.7 Documentation

In the Operational Guide:

- Document how to import a Solution.
- Document how to deploy a Solution.
- Document how to upgrade/downgrade a Solution.
- Document how to delete a Solution.

In the Developer Guide:

- Document how to monitor a Solution (ServiceMonitor, Service, ...).
- Document how to interface with the identity provider (Dex).
- Document how to build a Solution (layout, how to package, ...).

20.6.8 Test Plan

First of all, we must develop a Solution example, with at least 2 different versions, to be able to test the whole feature.

Then, we need to develop automated tests to ensure feature is working as expected. The tests will have to cover the following points:

- Solution installation and lifecycle (through both UI & CLI):
 - Importing / removing a Solution archive
 - Activating / deactivating a Solution
 - Creating / deleting an Environment
 - Adding / removing a Solution in / from an Environment
 - Upgrading / downgrading a Solution
- Solution can be plugged to MetalK8s cluster services (monitoring, alerting, ...).

20.7 Continuous Testing

This document will not describe how to write a test but just the list of tests that should be done and when.

The goal is to:

- have day-to-day development and PRs merged faster
- have a great test coverage

Lets define 2 differents stages of continuous testing:

- Pre-merge: Run during development process on changes not yet merged
- Post-merge: Run on changes already approved and merged in development branches

20.7.1 Pre-merge

What should be tested in pre-merge on every branch used during development (user/*, feature/*, improvement/*, bugfix/*, w/*). The pre-merge test should not long too much time (less than 40 minutes is great) so we can't test everything in pre-merge, we should only test building of the product and check that product still usable.

- Building tests
 - Build
 - Lint
 - Unit tests
- Installation tests
 - Simple install RHEL
 - Simple install CentOS + expansion

When merging several pull requests at the same time, given that we are on a queue branch (q/*), we may require additional tests as a combination of several PRs could have a larger impact than all individual PR:

- Simple upgrade/downgrade

20.7.2 Post-merge

On each and every development/2.* branches we want to run complex tests, that take more time or need more resources than classic tests that run during pre-merge, to ensure that the current product continues to work well.

Nightly

- Upgrade, downgrade tests:
 - For previous development branch

e.g.: on development/2.x test upgrade from development/2.(x-1) and downgrade to development/2.(x-1)

 - * Build branch development/2.(x-1) (or retrieve it if available)
 - * Tests:
 - Single node test
 - Complex deployment test
 - For last released version of current minor

e.g.: on development/2.x when developing “2.x.y-dev” test upgrade from metalk8s-2.x.(y-1) and downgrade to metalk8s-2.x.(y-1)

 - * Single node test
 - * Complex deployment test
 - For last released version of previous minor

e.g.: on development/2.x when developing “2.x.y-dev” test upgrade from metalk8s-2.(x-1).z and downgrade to metalk8s-2.(x-1).z where “2.(x-1).z” is the last patch released version for “2.(x-1)” (z may be different from y)

 - * Single node test
 - * Complex deployment test
- Backup, restore tests:
 - Environment with at least 3-node etcd cluster, destroy the bootstrap node and spawning a new fresh node for restoration
 - Environment with at least 3-node etcd cluster, destroy the bootstrap node and use one existing node for restoration
- Solutions tests

Note: Complex deployment is (to be validated):

- 1 bootstrap
 - 1 etcd and control
 - 1 etcd and control and workload
 - 1 workload and infra
 - 1 workload
 - 1 infra
-

Todo:

- Describe solutions tests (#1993)
-

Weekly

More complex tests:

- Performance/conformance tests
- Validation of *contrib* tooling (Heat, terraform, ...)
- Installation of “real” Solution (Zenko, ...)
- Long lifecycle metalk8s tests (several upgrade, downgrade, backup/restore, expansions, ...)

Todo: Validate the list of Weekly test to do and define exactly what need to be tested

20.7.3 Adaptive test plan

CI pre-merge may be more flexible by including some logic about the content of the changeset.

The goal here is to test only what needed according to the content of the commit.

For example:

- For a commit that changes uniquely documentation, we don’t need to run the entire installation test suite but rather run tests related to documentation.
- For a commit touching upgrade orchestrate we want to test upgrade directly in pre-merge and not wait *Post merge* build to get the test result.

Todo: Several questions:

- How to get the change of one commit ?
 - Depending on the files changed
 - * How do you know when you change something in salt if this changeset touch upgrade for example ?
 - ...
 - A tag in the commit message
 - * maybe ?
 - How to get the bunch of commit to test ?
 - Get commit between HEAD and target branch
 - * How to get this target ?
 - ...
-

DESIGN DOCUMENTS

21.1 Volume Management v1.0

- MetalK8s-Version: 2.4
- Replaces:
- Superseded-By:

21.1.1 Abstract

To be able to run stateful services (such as Prometheus, Zenko or Hyperdrive), MetalK8s needs the ability to provide and manage persistent storage resources.

To do so we introduce the concept of MetalK8s **Volume**, using a **Custom Resource Definition** (CRD), built on top of the existing concept of **Persistent Volume** from Kubernetes. Those **Custom Resources** (CR) will be managed by a dedicated Kubernetes operator which will be responsible for the storage preparation (using Salt states) and lifetime management of the backing **Persistent Volume**.

Volume management will be available from the Platform UI (through a dedicated tab under the Node page). There, users will be able to create, monitor and delete MetalK8s volumes.

21.1.2 Scope

The scope of this first version of Volume Management will be minimalist but still functionally useful.

Goals

- support two kinds of **Volume**:
 - **sparseLoopDevice** (backed by a sparse file)
 - **rawBlockDevice** (using whole disk)
- add support for volume creation (one by one) in the Platform UI
- add support for volume deletion (one by one) in the Platform UI
- add support for volume listing/monitoring (show status, size, ...) in the Platform UI
- document how to create a volume
- document how to create a **StorageClass** object
- automated tests on volume workflow (creation, deletion, ...)

Non-Goals

- RAID support
- LVM support
- expose raw block device (unformatted) as **Volume**
- use an **Admission Controller** for semantic validation
- auto-discovery of the disks
- batch provisioning from the Platform UI

21.1.3 Proposal

To implement this feature we need to:

- define and deploy a new CRD describing a MetalK8s **Volume**
- develop and deploy a new Kubernetes operator to manage the MetalK8s volumes
- develop new Salt states to prepare and cleanup underlying storage on the nodes
- update the Platform UI to allow volume management

User Stories

Volume Creation

As a user I need to be able to create MetalK8s volume from the Platform UI.

At creation time I can specify the type of volume I want, and then either its size (for **sparseLoopDevice**) or the backing device (for **rawBlockDevice**).

I should be able monitor the progress of the volume creation from the Platform UI and see when the volume is ready to use (or if an error occurred).

Volume Monitoring

As a user I should be able to see all the volumes existing on a specified node as well as their states.

Volume Deletion

As a user I need to be able to delete a MetalK8s volume from the Platform UI when I no longer need it.

The Platform UI should prevent me from deleting Volumes in use.

I should be able monitor the progress of the volume deletion from the Platform UI.

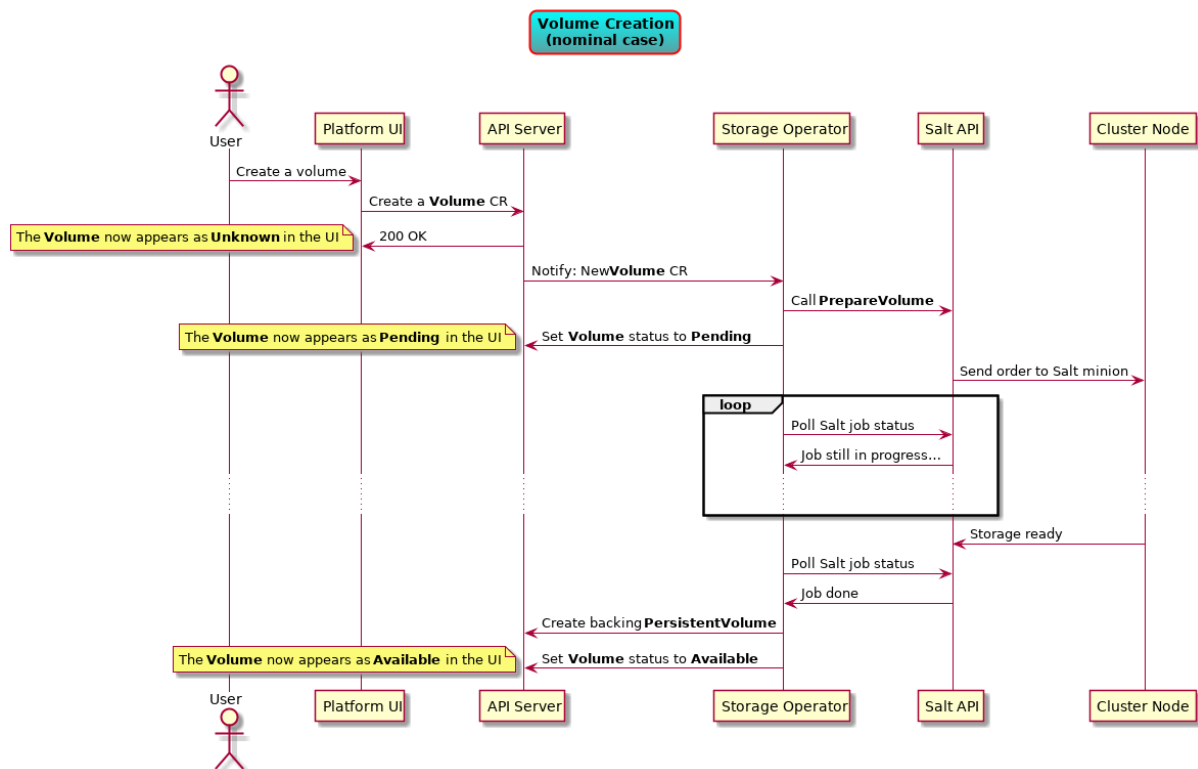
Component Interactions

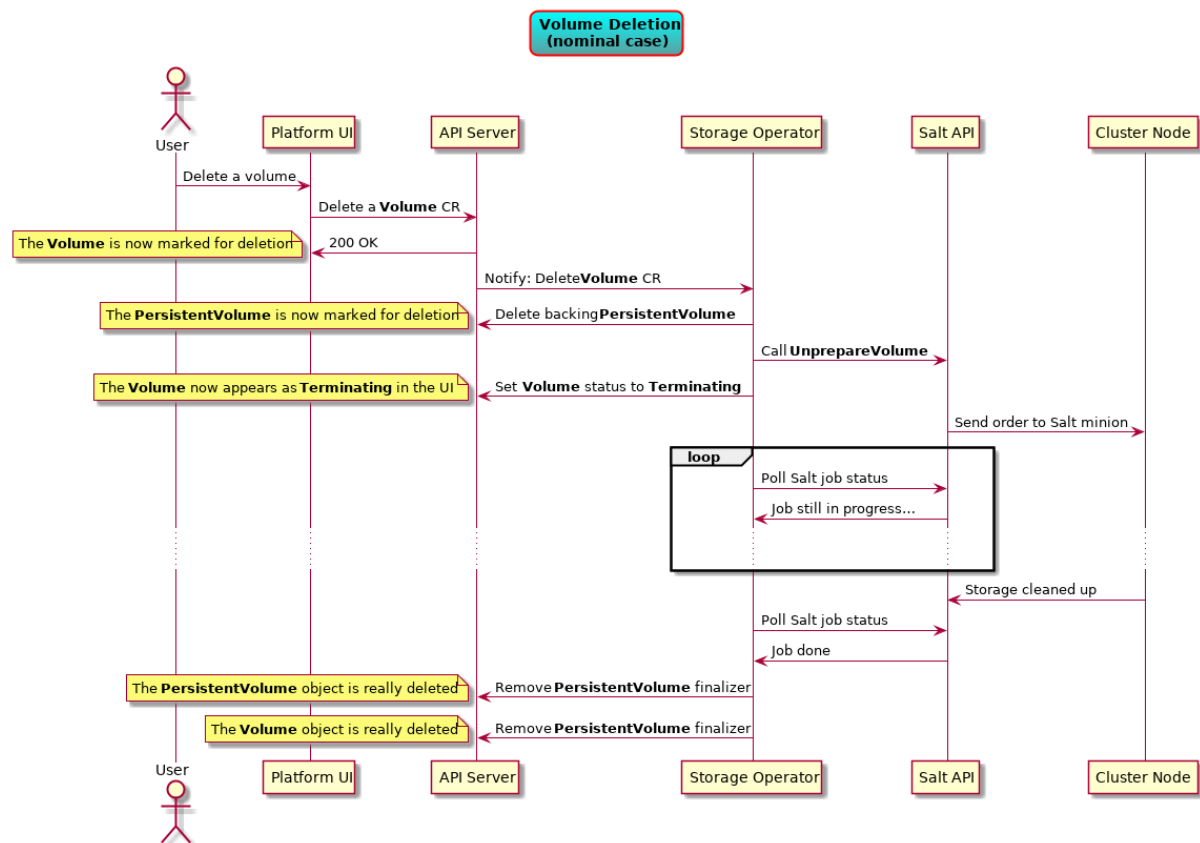
User will create MetalK8s volumes through the Platform UI.

The Platform UI will create and delete **Volume** CRs from the API server.

The operator will watch events related to **Volume** CRs and **PersistentVolume** CRs owned by a **Volume** and react in order to update the state of the cluster to meet the desired state (prepare storage when a new **Volume** CR is created, clean up resources when a **Volume** CR is deleted). It will also be responsible for updating the states of the volumes.

To do its job, the operator will rely on Salt states that will be called asynchronously (to avoid blocking the reconciliation loop and keep a reactive system) through the Salt API. Authentication to the Salt API will be done through a dedicated Salt account (with limited privileges) using credentials from a dedicated cluster **Service Account**.





21.1.4 Implementation Details

Volume Status

A **PersistentVolume** from Kubernetes has the following states:

- **Pending:** used for **PersistentVolume** that is not available
- **Available:** a free resource that is not yet bound to a claim
- **Bound:** the volume is bound to a claim
- **Released:** the claim has been deleted, but the resource is not yet reclaimed by the cluster
- **Failed:** the volume has failed its automatic reclamation

Similarly, our **Volume** object will have the following states:

- **Available:** the backing storage is ready and the associated **PersistentVolume** was created
- **Pending:** preparation of the backing storage in progress (e.g. an asynchronous Salt call is still running).
- **Failed:** something is wrong with the volume (Salt state execution failed, invalid value in the CRD, ...)
- **Terminating:** cleanup of the backing storage in progress (e.g. an asynchronous Salt call is still running).

Operator Reconciliation Loop

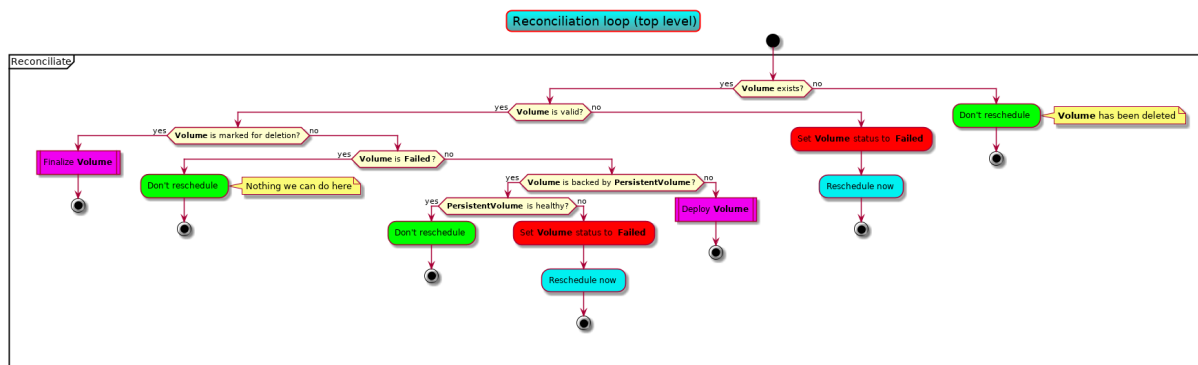
Reconciliation Loop (Top Level)

When the operator receives a request, the first thing it does is to fetch the targeted **Volume**. If it doesn't exist, which happens when a volume is **Terminating** and has no finalizer, then there's nothing more to do.

If the volume does exist, the operator has to check its semantic validity.

Once pre-checks are done, there are four cases:

1. the volume is marked for deletion: the operator will try to delete the volume (more details in [Volume Finalization](#)).
2. the volume is stuck in an unrecoverable (automatically at least) error state: the operator can't do anything here, the request is considered done and won't be rescheduled.
3. the volume doesn't have a backing **PersistentVolume** (e.g. newly created volume): the operator will deploy the volume (more details in [Volume Deployment](#)).
4. the backing **PersistentVolume** exists: the operator will check its status to update the volume's status accordingly.



Volume Deployment

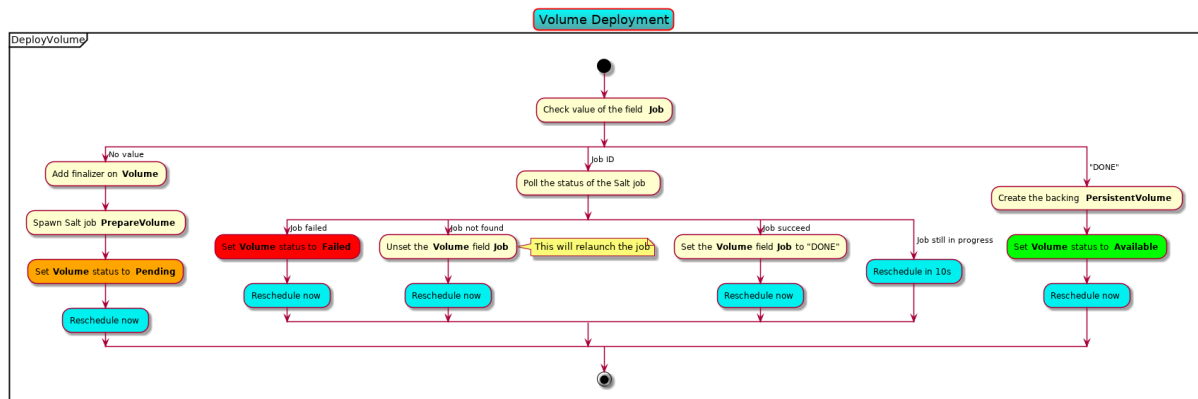
To deploy a volume, the operator needs to prepare its storage (using Salt) and create a backing **PersistentVolume**.

If the **Volume** object has no value in its Job field, it means that the deployment hasn't started, thus the operator will set a finalizer on the **Volume** object and then start the preparation of the storage using an asynchronous Salt call (which gives a job ID) before rescheduling the request to monitor the evolution of the job.

If the **Volume** object has a job ID, then the storage preparation is in progress and the operator will monitor it until it's over. If the Salt job ends with an error, the operator will move the volume into a failed state.

Otherwise (i.e. Salt job succeeded), the operator will proceed with the **PersistentVolume creation** (which requires an extra Salt call, synchronous this time, to get the volume size), taking care of putting a finalizer on the **PersistentVolume** (so that its lifetime is tied to the **Volume**'s) and set the **Volume** as the owner of the created **PersistentVolume**.

Once the **PersistentVolume** is successfully created, the operator will move the **Volume** to the *Available* state and reschedule the request (the next iteration will check the health of the **PersistentVolume** just created).

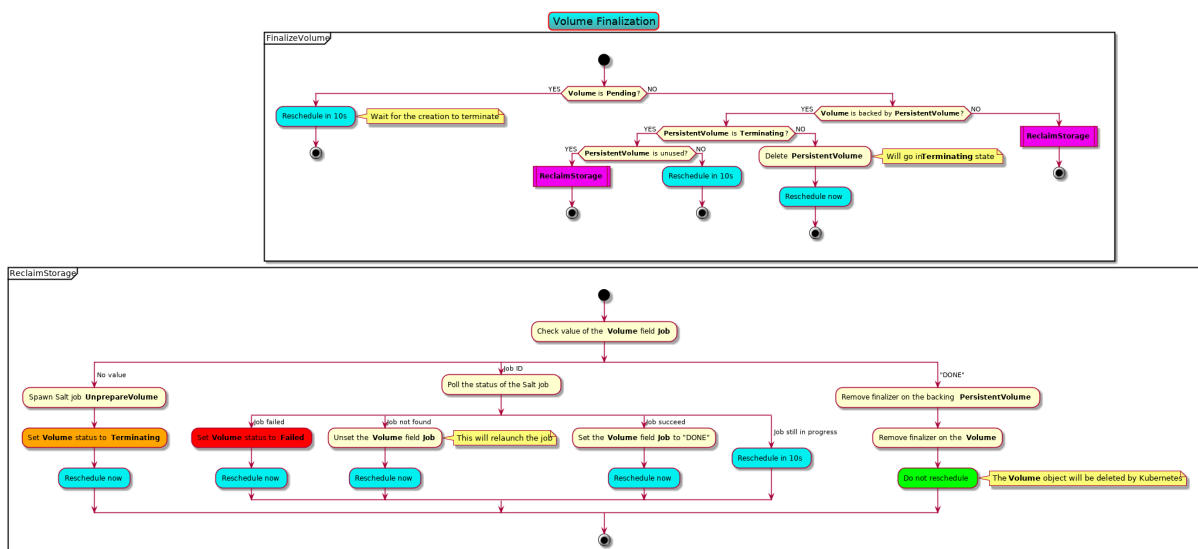


Volume Finalization

A **Volume** in state **Pending** cannot be deleted (because the operator doesn't know where it is in the creation process). In such cases, the operator will reschedule the request until the volume becomes either **Failed** or **Available**.

For volumes with no backing **PersistentVolume**, the operator will directly reclaim the storage on the node (using an asynchronous Salt job) and upon completion it will remove the **Volume** finalizer to let Kubernetes delete the object.

If there is a backing **PersistentVolume**, the operator will delete it (if it's not already in a terminating state) and watch for the moment when it becomes unused (this is done by rescheduling). Once the backing **PersistentVolume** becomes unused, the operator will reclaim its storage and remove the finalizers to let the object be deleted.



Volume Deletion Criteria

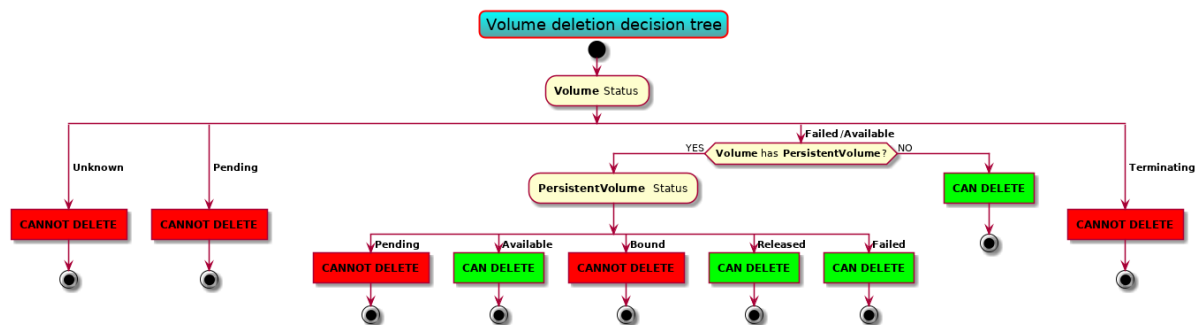
A volume should be deletable from the UI when it's deletable from a user point of view (you can always delete an object from the API), i.e. when deleting the object will trigger an “immediate” deletion (i.e. the object won't be retained).

Here are the few rules that are followed to decide if a **Volume** can be deleted or not:

- **Pending** states are left untouched: we wait for the completion of the pending action before deciding which action to take.
- The lack of status information is a transient state (can happen between the **Volume** creation and the first iteration of the reconciliation loop) and thus we make no decision while the status is unset.
- **Volume** objects whose **PersistentVolume** is bound cannot be deleted.
- **Volume** objects in **Terminating** state cannot be deleted because their deletion is already in progress!

In the end, a **Volume** can be deleted in two cases:

- it has no backing **PersistentVolume**
- the backing **PersistentVolume** is not bound (**Available**, **Released** or **Failed**)



21.1.5 Documentation

In the Operational Guide:

- document how to create a volume from the CLI
- document how to delete a volume from the CLI
- document how to create a volume from the UI
- document how to delete a volume from the UI
- document how to create a **StorageClass** from the CLI (and mention that we should set **Volume-BindingMode** to **WaitForFirstConsumer**)

In the Developer Documentation:

- document how to run the operator locally
- document this design

21.1.6 Test Plan

We should have automated end-to-end tests of the feature (creation and deletion), from the CLI and maybe on the UI part as well.

HOW TO BUILD METALK8S

22.1 Requirements

In order to build MetalK8s we rely and third-party tools, some of them are mandatory, others are optional.

22.1.1 Mandatory

- [Python](#) 3.6 or higher: our buildchain is Python-based
- [docker](#) 17.03 or higher: to build some images locally
- [skopeo](#), 0.1.19 or higher: to save local and remote images
- [hardlink](#): to de-duplicate images layers
- [mkisofs](#): to create the MetalK8s ISO

22.1.2 Optional

- [git](#): to add the Git reference in the build metadata
- [Vagrant](#), 1.8 or higher: to spawn a local cluster (VirtualBox is currently the only provider supported)
- [VirtualBox](#): to spawn a local cluster
- [tox](#): to run the linters

22.1.3 Development

If you want to develop on the buildchain, you can add the development dependencies with `pip install -r requirements/build-dev-requirements.txt`.

22.2 How to build an ISO

Our build system is based on [doit](#).

To build, simply type `./doit.sh`.

Note that:

- you can speed up the build by spawning more workers, e.g. `./doit.sh -n 4`.
- you can have a JSON output with `./doit.sh --reporter json`

When a task is prefixed by:

- `--`: the task is skipped because already up-to-date
- `..`: the task is executed
- `!!`: the task is ignored.

22.2.1 Main tasks

To get a list of the available targets, you can run `./doit.sh list`.

The most important ones are:

- `iso`: build the MetalK8s ISO
- `lint`: run the linting tools on the codebase
- `populate_iso`: populate the ISO file tree
- `vagrant_up`: spawn a development environment using Vagrant

By default, i.e. if you only type `./doit.sh` with no arguments, the `iso` task is executed.

You can also run a subset of the build only:

- `packaging`: download and build the software packages and repositories
- `images`: download and build the container images
- `salt_tree`: deploy the Salt tree inside the ISO

22.3 Configuration

You can override some buildchain's settings through a `.env` file at the root of the repository.

Available options are:

- `PROJECT_NAME`: name of the project
- `BUILD_ROOT`: path to the build root (either absolute or relative to the repository)
- `VAGRANT_PROVIDER`: type of machine to spawn with Vagrant
- `VAGRANT_UP_ARGS`: command line arguments to pass to `vagrant up`
- `VAGRANT_SNAPSHOT_NAME`: name of auto generated Vagrant snapshot
- `DOCKER_BIN`: Docker binary (name or path to the binary)
- `GIT_BIN`: Git binary (name or path to the binary)
- `HARDLINK_BIN`: hardlink binary (name or path to the binary)
- `MKISOFS_BIN`: mkisofs binary (name or path to the binary)
- `SKOPEO_BIN`: skopeo binary (name or path to the binary)
- `VAGRANT_BIN`: Vagrant binary (name or path to the binary)
- `GOFMT_BIN`: gofmt binary (name or path to the binary)
- `OPERATOR_SDK_BIN`: the Operator SDK binary (name or path to the binary)

Default settings are equivalent to the following `.env`:

```
export PROJECT_NAME=MetalK8s
export BUILD_ROOT=_build
export VAGRANT_PROVIDER=virtualbox
export VAGRANT_UP_ARGS="--provision --no-destroy-on-error --parallel --provider $VAGRANT_PROVIDER"
export DOCKER_BIN=docker
```

(continues on next page)

(continued from previous page)

```
export HARDLINK_BIN=hardlink
export GIT_BIN=git
export MKISOFS_BIN=mkisofs
export SKOPEO_BIN=skopeo
export VAGRANT_BIN=vagrant
export GOFMT_BIN=gofmt
export OPERATOR_SDK_BIN=operator-sdk
```

22.4 Buildchain features

Here are some useful doit commands/features, for more information, [the official documentation is here](#).

22.4.1 doit tabcompletion

This generates completion for bash or zsh (to use it with your shell, see [the instructions here](#)).

22.4.2 doit list

By default, `./doit.sh list` only shows the “public” tasks.

If you want to see the subtasks as well, you can use the option `--all`.

```
% ./doit.sh list --all
images      Pull/Build the container images.
iso         Build the MetalK8s image.
lint        Run the linting tools.
lint:shell  Run shell scripts linting.
lint:yaml   Run YAML linting.
[...]
```

Useful if you only want to run a part of a task (e.g. running the lint tool only on the YAML files).

You can also display the internal (a.k.a. “private” or “hidden”) tasks with the `-p` (or `--private`) options.

And if you want to see **all** the tasks, you can combine both: `./doit.sh list --all --private`.

22.4.3 doit clean

You can cleanup the build tree with the `./doit.sh clean` command.

Note that you can have fine-grained cleaning, i.e. cleaning only the result of a single task, instead of trashing the whole build tree: e.g. if you want to delete the container images, you can run `./doit.sh clean images`.

You can also execute a dry-run to see what would be deleted by a clean command: `./doit.sh clean -n images`.

22.4.4 doit info

Useful to understand how tasks interact with each others (and for troubleshooting), the `info` command display the task's metadata.

Example:

```
% ./doit.sh info _build_rpm_packages:calico-cni-plugin/srpm

_build_rpm_packages:calico-cni-plugin/srpm

Build calico-cni-plugin-3.8.2-1.el7.src.rpm

status      : up-to-date

file_dep    :
- /home/foo/dev/metalk8s/_build/packages/redhat/calico-cni-plugin/SOURCES/calico-ipam-amd64
- /home/foo/dev/metalk8s/_build/packages/redhat/calico-cni-plugin/SOURCES/v3.8.2.tar.gz
- /home/foo/dev/metalk8s/packages/redhat/calico-cni-plugin.spec
- /home/foo/dev/metalk8s/_build/packages/redhat/calico-cni-plugin/SOURCES/calico-amd64

task_dep    :
- _package_mkdir_rpm_root
- _build_builder:metalk8s-rpm-builder
- _build_rpm_packages:calico-cni-plugin/mkdir

targets     :
- /home/foo/dev/metalk8s/_build/packages/redhat/calico-cni-plugin-3.8.2-1.el7.src.rpm
```

22.4.5 Wildcard selection

You can use wildcard in task names, which allows you to either:

- execute all the sub-tasks of a specific task: `_build_rpm_packages:calico-cni-plugin/*` will execute all the tasks required to build the package.
- execute a specific sub-task for all the tasks: `_build_rpm_packages:*/get_source` will retrieve the source files for all the packages.

HOW TO RUN COMPONENTS LOCALLY

23.1 Running a cluster locally

23.1.1 Requirements

- the *mandatory requirements for the buildchain*
- **Vagrant**, 1.8 or higher: to spawn a local cluster (VirtualBox is currently the only provider supported)
- **VirtualBox**: to spawn a local cluster

23.1.2 Procedure

You can spawn a local MetalK8s cluster by running `./doit.sh vagrant_up`.

This command will start a virtual machine (using VirtualBox) and:

- mount the build tree
- import a private SSH key (automatically generated in `.vagrant`)
- generate a bootstrap configuration
- execute the bootstrap script to make this machine a bootstrap node
- provision sparse-file Volumes for Prometheus and Alertmanager to run on this bootstrap node

After executing this command, you have a MetalK8s bootstrap node up and running and you can connect to it by using `vagrant ssh bootstrap`.

Note that you can extend your cluster by spawning extra nodes (up to 9 are already pre-defined in the provided Vagrantfile) by running `vagrant up node1 --provision`. This will:

- spawn a virtual machine for the node 1
- import the pre-shared SSH key into it

You can then follow the cluster expansion procedure to add the freshly spawned node into your MetalK8s cluster (you can get the node's IP with `vagrant ssh node1 -- sudo ip a show eth1`).

23.2 Running the storage operator locally

23.2.1 Requirements

- [Go](#) (1.12 or higher) and [operator-sdk](#) (0.9 or higher): to build the Kubernetes Operators
- [Mercurial](#): some Go dependencies are downloaded from Mercurial repositories.

23.2.2 Prerequisites

- You should have a running MetalK8s cluster somewhere
- You should have installed the dependencies locally with `cd storage-operator; go mod download`

23.2.3 Procedure

1. Copy the `/etc/kubernetes/admin.conf` from the bootstrap node of your cluster onto your local machine
2. Delete the already running storage operator, if any, with `kubectl --kubeconfig /etc/kubernetes/admin.conf -n kube-system delete deployment storage-operator`
3. Get the address of the Salt API server with `kubectl --kubeconfig /etc/kubernetes/admin.conf -n kube-system describe svc salt-master | grep :4507`
4. Run the storage operator with:

```
cd storage-operator
export KUBECONFIG=<path-to-the-admin.conf-you-copied-locally>
export METALK8S_SALT_MASTER_ADDRESS=https://<ADDRESS-OF-SALT-API>
operator-sdk up local
```

23.3 Running the platform UI locally

23.3.1 Requirements

- [Node.js](#), 10.16

23.3.2 Prerequisites

- You should have a running MetalK8s cluster somewhere
- You should have installed the dependencies locally with `cd ui; npm install`

23.3.3 Procedure

1. Connect to the bootstrap node of your cluster, and execute the following command as root:

```
python - <<EOF
import subprocess
import json

output = subprocess.check_output([
    'salt-call', 'pillar.get', 'metalk8s', '--out', 'json'
])
```

(continues on next page)

(continued from previous page)

```
pillar = json.loads(output)['local']
output = subprocess.check_output([
    'salt-call', 'grains.get', 'metalk8s:control_plane_ip', '--out', 'json'
])
control_plane_ip = json.loads(output)['local']
ui_conf = {
    'url': 'https://{ip}:6443'.format(control_plane_ip),
    'url_salt': 'https://{salt[ip]}:{salt[ports][api]}'.format(
        salt=pillar['endpoints']['salt-master']
    ),
    'url_prometheus': 'http://{prom[ip]}:{prom[ports][web][node_port]}'.format(
        prom=pillar['endpoints']['prometheus']
    ),
}
print(json.dumps(ui_conf, indent=4))
EOF
```

2. Copy the output into ui/public/config.json.
3. Run the UI with `cd ui; npm run start`

24.1 Continuous Testing

24.1.1 Add a new test in the continuous integration system

When we refer to test, at continuous integration system level, it means an end-to-end task (building, linting, testing, ...) that requires a dedicated environment, with one or several machines (virtual or container).

A test that only checks a specific feature of a classic MetalK8s deployment should be part of PyTest BDD and not integrated as a dedicated stage in continuous integration system (e.g.: Testing that Ingress Pod are running and ready is a feature of MetalK8s that should be tested in PyTest BDD and not directly as a stage in continuous integration system).

How to choose between Pre-merge and Post-merge

The choice really depends on the goals of this test.

As a high-level view:

Pre-merge:

- Test is usually not long and could last less than 30 minutes.
- Test essential features of the product (installation, expansion, building, ...).

Post-merge:

- Test last longer (more than 30 minutes).
- Test “non-essential” (not mandatory to have a working cluster) feature of the product (upgrade, downgrade, solutions, ...).

How to add a stage in continuous integration system

Continuous integration system is controlled by the `eve/main.yml` YAML file.

A stage is defined by a worker and a list of steps. Each stage should be in the `stages` section and triggered by `pre-merge` or `post-merge`.

To know the different kind of workers available, all the builtin steps, how to trigger a stage, ... please refer to the eve documentation.

A test stage in MetalK8s context

In MetalK8s context each test stage (eve stage that represents a full test) should generate a status file containing the result of the test, either a success or a failure, and a JUnit file containing the result of the test and information about this test.

To generate the JUnit file, each stage needs the following information:

- The name of the Test Suite this test stage is part of
- Section path to group tests in a Test Suite if needed (optional)
- A test name

Before executing all the steps of the test we first generate a failed result and at the end of the test we generate a success result. So that the failed result get overridden by the success one if everything goes well.

At the very end, the final status of a test should be uploaded no matter the outcome of the test.

To generate these results, we already have several helpers available.

Example:

Consider we want a new test named `My Test` which is part of the subsection `My sub section` of the section `My section` in the test suite `My Test Suite`.

Note: Test, suite and class names are not case sensitive in `eve/main.yml`.

```
my-stage:
  _metalk8s_internal_info:
    junit_info: &_my_stage_junit_info
    TEST_SUITE: my test suite
    CLASS_NAME: my section.my sub section
    TEST_NAME: my test
  worker:
    # ...
    # Worker informations
    # ...
  steps:
    - Git: *git_pull
    - ShellCommand: # Generate a failed final status
      <<: *add_final_status_artifact_failed
      env:
        <<: *_env_final_status_artifact_failed
        <<: *_my_stage_junit_info
        STEP_NAME: my-stage
    # ...
    # All test steps should be here !
    # ...
    - ShellCommand: # Generate a success final status
      <<: *add_final_status_artifact_success
      env:
        <<: *_env_final_status_artifact_success
        <<: *_my_stage_junit_info
        STEP_NAME: my-stage
    - Upload: *upload_final_status_artifact
```


TestRail upload

To store results, we use TestRail which is a declarative engine. It means that all test suites, plans, cases, runs, etc. must be declared, before proceeding to the results upload.

Warning: TestRail upload is only done for Post-merge as we do not want to store each and every test result coming from branches with on-going work.

Do not follow this section if it's not a Post-merge test stage.

The file `eve/testrail_description_file.yaml` contains all the TestRail object declarations, that will be created automatically during Post-merge stage execution.

It's a YAML file used by TestRail UI to describe the objects.

Example:

```
My Test Suite:
  description: >-
    My first test suite description
  section:
    My Section:
      description: >-
        My first section description
      sub_sections:
        My sub section:
          description: >-
            My first sub section description
          cases:
            My test: {}
        # sub_sections: <-- subsections can be nested as deep as needed
```

24.2 Commit Best Practices

24.2.1 How to split a change into commits

Why do we need to split changes into commits

This has several advantages amongst which are:

- small commits are easier to review (a large pull request correctly divided into commits is easier/faster to review than a medium-sized one with less thought-out division)
- simple commits are easier to revert ([e866b01f0553/8208a170ac66](#))/cherry-pick ([Pull request #1641](#))
- when looking for a regression (e.g. using `git bisect`) it is easier to find the root cause
- make `git log` and `git blame` way more useful

Examples

The golden rule to create good commits is to ensure that there is only one “logical” change per commit.

Cosmetic changes

Use a dedicated commit when you want to make cosmetic changes to the code (linting, whitespaces, alignment, renaming, etc.).

Mixing cosmetics and functional changes is bad because the cosmetics (which tend to generate a lot of diff/noise) will obscure the important functional changes, making it harder to correctly determine whether the change is correct during the review.

Example (Pull request [#1620](#)):

- one commit for the cosmetic changes: [766f572e462c6933c8168a629ed4f479bb68a803](#)
- one commit for the functional changes: [3367fabdefc0b35d34bf7cf2fb0d33ff81f9fd5a](#)

Ideally, purely cosmetic changes which inflate the number of changes in a PR significantly, should go in a separate PR

Refactoring

When introducing new features, you often have to add new helpers or refactor existing code. In such case, instead of having single commit with everything inside, you can either:

1. first add a new helper: [29f49cbe9dfa](#)
2. then use it in new code: [7e47310a8f20](#)

Or:

1. first add the new code: [5b2a6d5fa498](#)
2. then refactor the now duplicated code: [ac08d0f53a83](#)

Mixing unrelated changes

It is sometimes tempting to do small unrelated changes as you are working on something else in the same code area. Please refrain to do so, or at least do it in a dedicated commit.

Mixing non-related changes into the same commit makes revert and cherry-pick harder (and understanding as well).

The pull request [#1846](#) is a good example. It tackles three issues at once: [#1830](#) and [#1831](#) (because they are similar) and [#839](#) (because it was making the other changes easier), but it uses distinct commits for each issue.

24.2.2 How to write a commit message

Why do we need commit messages

After comments in the code, commit messages are the easiest way to find context for every single line of code: running `git blame` on a file will give you, for each line, the identifier of the last commit that changed the line.

Unlike a comment in the code (which applies to a single line or file), a commit message applies to a logical change and thus can provide information on the design of the code and why the change was done. This makes commit messages a part of the code documentation and makes them helpful for other developers to understand your code.

Last but not least: commit messages can also be used for automating tasks such as issue management.

Note that it is important to have all the necessary information in the commit message, instead of having them (only) in the related issue, because:

- the issue can contain troubleshooting/design discussion/investigation with a lot of back and forth, which makes hard to get the gist of it.
- you need access to an external service to get the whole context, which goes against one of biggest advantage of the distributed SCM (having all the information you need offline, from your local copy of the repository).
- migration from one tracking system to another will invalidate the references/links to the issues.

Anatomy of a good commit message

A commit is composed of a subject, a body and a footer. A blank line separates the subject from body and the body from the footer.

The body can be omitted for trivial commit. That being said, be very careful: a change might seem trivial when you write it but will seem totally awkward the day you will have to understand why you made it. If you think your patch is trivial and somebody tells you he does not understand your patch, then your patch is not trivial and it requires a detailed description.

The footer contains references for issue management (Refs, Closes, etc.) or other relevant annotations (cherry-pick source, etc.). Optional if your commit is not related to any issue (should be pretty rare).

Subject

A good commit message should start with a short summary of the change: the subject line.

This summary should be written using the imperative mood and carry as much information as possible while staying short, ideally under 50 characters (this is a goal, the hard limit is 72).

Subject topic and description shouldn't start with a capital.

It is composed of:

- a topic, usually the name of the affected component (ui, build, docs, etc.)
- a slash and then the name of the sub-component (optional)
- a colon
- the description of the change

Examples:

- ci: use proxy-cache to reduce flakiness
- build/package: factorize task_dep in DEBPackage
- ui/volume: add banner when failed to create volume

If several components are affected:

- split your commit (preferred)
- pick only the most affected one
- entirely omit the component (happen for truly global change, like renaming licence to license over the whole codebase)

As for “what is the topic?”, the following heuristic works quite well for MetalK8s: take the name of the top-level directory (ui, salt, docs, etc.) except for eve (use ci instead). buildchain could also be shortened to build.

Having the topic in the summary line allows for faster peering over `git log` output (you can know what the commit is about just by reading a few characters, not need to check the entire commit message or the associated diff). It also helps the review process: if you have a big pull request affecting front-end and back-end, front-end people can only review commits starting with `ui` (not need to read over the whole diff, or to open each commit one by one in Github to see which ones are interesting).

Body

The body should answer the following questions:

- Why did you make this change? (is this for a new feature, a bugfix - then, why was it buggy? -, some cleanup, some optimization, etc.). It is really important to describe the intent/motivation behind the changes.
- What change did you make? Document what the original problem was and how it is being fixed (can be omitted for short obvious patches).
- Why did you make the change in that way and not in another (mention alternate solutions considered but discarded, if any)?

When writing your message you must consider that your reader does not know anything about the code you have patched.

You should also describe any limitations of the current code. This will avoid reviewer pointing them out, and also inform future people looking at the code which tradeoffs were made at the time.

Lines must be wrapped at 72 characters.

Footer

Use [references](#) such as Refs, See, Fixes or Closes followed by an issue number to automate issue management.

In addition to the references, you can also provide the URLs (it will be quicker to access them from the terminal).

Example:

```
topic: description

[ commit message body ]

Refs: #XXXXX
Refs: #YYYYY
Closes: #ZZZZZ
See: https://github.com/scality/metalk8s/issues/XXXXX
See: https://github.com/scality/metalk8s/issues/YYYYY
See: https://github.com/scality/metalk8s/issues/ZZZZZ
```

Footer can also contain a signature (`git commit -s`) or cherry-pick source (`git cherry-pick -x`).

Examples

Bad commit message

- Quick fix for service port issue: what was the issue? It is a quick fix, why not a proper fix? What are the limitations?
- fix glitches: as expressive and useful as ~fix stuff~
- Bump Create React App to v3 and add optional-chaining: Why? What are the benefits?
- Add skopeo & m2crypto to packages list: Why do we need them?
- Split certificates bootstrap between CA and clients: Why do we need this split? What is the issue we are trying to solve here?

Note that none of these commits contain a reference to an issue (which could have been used as an (invalid) excuse for the lack of information): you really have no more context/explanation than what is shown here.

Good commit message

Commit b531290c04c4

Add gzip to nginx conf

This will decrease the size of the file the client need to download
In the current version we have ~7x improvement.
From 3.17Mb to 0.470Mb send to the client

Some things to note about this commit message:

- Reason behind the changes are explained: we want to decrease the size of the downloaded resources.
- Results/effects are demonstrated: measurements are given.

Commit 82d92836d4ff

Use safer invocation of shell commands

Running commands with the "host" fixture provided by testinfra was done without concern for quoting of arguments, and might be vulnerable to injections / escaping issues.

Using a log-like formatting, i.e. ``host.run('my-cmd %s %d', arg1, arg2)`` fixes the issue (note we cannot use a list of strings as with ``subprocess``).

Issue: GH-781

Some things to note about this commit message:

- Reasons behind the changes are explained: potential security issue.
- Solution is described: we use log-like formatting.
- Non-obvious parts are clarified: cannot use a list of string (as expected) because it is not supported.

Commit f66ac0be1c19

build: fix concurrent build on MacOS

When trying to use the parallel execution feature of ``doit`` on Mac, we observe that the worker processes are killed by the OS and only the main one survives.

The issues seems related to the fact that:

- by default ``doit`` uses ``fork`` (through ``multiprocessing``) to spawn its workers
- since macOS 10.13 (High Sierra), Apple added a new security measure[1] that kill processes that are using a dangerous mix of threads and forks[2])

As a consequence, now instead of working most of the time (and failing in a hard way to debug), the processes are directly killed.

There are three ways to solve this problems:

1. set the environment variable ``OBJC_DISABLE_INITIALIZE_FORK_SAFETY=YES.``
2. don't use ``fork``
3. fix the code that uses a dangerous mix of thread and forks

(1) is not good as it doesn't fix the underlying issue: it only disable the security and we're back to "works most of the time, sometimes does weird things"

(2) is easy to do because we can tell to ``doit`` to uses only threads instead of forks.

(3) is probably the best, but requires more troubleshooting/time/

In conclusion, this commit implements (2) until (3) is done (if ever) by detecting macOS and forcing the use of threads in that case.

[1]: http://sealiesoftware.com/blog/archive/2017/6/5/Objective-C_and_fork_in_macOS_1013.html

[2]: <https://blog.phusion.nl/2017/10/13/why-ruby-app-servers-break-on-macos-high-sierra-and-what-can-be-done-about-it/>

Closes: #1354

Some things to note about this commit message:

- Observed problem is described: parallel builds crash on macOS.
- Root cause is analyzed: OS security measure + thread/fork mix.
- Several solution are proposed: disable the security, workaround the problem or fix the root cause.
- Selection of a solution is explained: we go for the workaround because it is easy and faster.
- Extra-references are given: links in the footer gives more in-depth explanations/context.

24.2.3 Conclusion

When reviewing a change, do not simply look at the correctness of the code: review the commit message itself and request improvements to its content. Look out for commits that can be divided, ensure that cosmetic changes are not mixed with functional changes, etc.

The goal here is to improve the long term maintainability, by a wide variety of developers who may only have the Git history to get some context so it is important to have a useful Git history.

24.3 Python best practices

24.3.1 Import

Avoid `from module_foo import symbol_bar`

In general, it is a good practice to avoid the form `from foo import bar` because it introduces two distinct bindings (`bar` is distinct from `foo.bar`) and when the binding in one namespace changes, the binding in the other will not. . .

That's also why this can interfere with the mocking.

All in all, this should be avoided when unnecessary.

Rationale

Reduce the likelihood of surprising behaviors and ease the mocking.

Example

```
# Good
import foo

baz = foo.Bar()

# Bad
from foo import Bar

baz = Bar()
```

References

- [Idioms and Anti-Idioms in Python](#)
- [unittest.mock documentation](#)

24.3.2 Naming

Predicate functions

Functions that return a Boolean value should have a name that starts with `has_`, `is_`, `was_`, `can_` or something similar that makes it clear that it returns a Boolean.

This recommendation also applies to Boolean variable.

Rationale

Makes code clearer and more expressive.

Example

```
class Foo:
    # Bad.
    def empty(self):
        return len(self.bar) == 0

    # Bad.
    def baz(self, initialized):
        if initialized:
            return
        # [...]

    # Good.
    def is_empty(self):
        return len(self.bar) == 0

    # Good.
    def qux(self, is_initialized):
        if is_initialized:
            return
        # [...]
```

24.3.3 Patterns and idioms

Don't write code vulnerable to "Time of check to time of use"

When there is a time window between the checking of a condition and the use of the result of that check where the result may become outdated, you should always follow the **EAFP** (It is Easier to Ask for Forgiveness than Permission) philosophy rather than the **LBYL** (Look Before You Leap) one (because it gives you a false sense of security).

Otherwise, your code will be vulnerable to the infamous **TOCTTOU** (Time Of Check To Time Of Use) bugs.

In Python terms:

- **LBYL**: if guard around the action
- **EAFP**: try/except statements around the action

Rationale

Avoid race conditions, which are a source of bugs and security issues.

Examples

```
# Bad: the file 'bar' can be deleted/created between the `os.access` and
# `open` call, leading to unwanted behavior.
if os.access('bar', os.R_OK):
    with open(bar) as fp:
        return fp.read()
return 'some default data'

# Good: no possible race here.
try:
    with open('bar') as fp:
        return fp.read()
except OSError:
    return 'some default data'
```

References

- Time of check to time of use

Minimize the amount of code in a try block

The size of a try block should be as small as possible.

Indeed, if the try block spans over several statements that can raise an exception caught by the except, it can be difficult to know which statement is at the origin of the error.

Of course, this rule doesn't apply to the catch-all try/except that is used to wrap existing exceptions or to log an error at the top level of a script.

Having several statements is also OK if each of them raises a different exception or if the exception carries enough information to make the distinction between the possible origins.

Rationale

Easier debugging, since the origin of the error will be easier to pinpoint.

Don't use `hasattr` in Python 2

To check the existence of an attribute, don't use `hasattr`: it shadows errors in properties, which can be surprising and hide the root cause of bugs/errors.

Rationale

Avoid surprising behavior and hard-to-track bugs.

Examples

```
# Bad.
if hasattr(x, "y"):
    print(x.y)
else:
    print("no y!")

# Good.
try:
    print(x.y)
except AttributeError:
    print("no y!")
```

References

- [hasattr\(\) – A Dangerous Misnomer](#)

INTEGRATING WITH METALK8S

25.1 Introduction

With a focus on having minimal human actions required, both in its deployment and operation, MetalK8s also intends to ease deployment and operation of complex applications, named *Solutions*, on its cluster.

This document defines what a *Solution* refers to, the responsibilities of each party in this integration, and will link to relevant documentation pages for detailed information.

25.1.1 What is a *Solution*?

We use the term *Solution* to describe a packaged Kubernetes application, archived as an ISO disk image, containing:

- A set of OCI images to inject in MetalK8s image registry
- An *Operator* to deploy on the cluster
- Optionally, a UI for managing and monitoring the application, represented by a standard Kubernetes Deployment

For more details, see the following documentation pages:

- [Solution archive guidelines](#)
- [Solution Operator guidelines](#)
- (TODO) Solution UI guidelines

Once a *Solution* is deployed on MetalK8s, a user can deploy one or more versions of the *Solution Operator*, using either the *Solution UI* or the Kubernetes API, into separate namespaces. Using the *Operator*-defined *CustomResource(s)*, the user can then effectively deploy the application packaged in the *Solution*.

25.1.2 How is a *Solution* declared in MetalK8s?

MetalK8s already uses a *BootstrapConfiguration* object, stored in `/etc/metalk8s/bootstrap.yaml`, to define how the cluster should be configured from the bootstrap node, and what versions of MetalK8s are available to the cluster.

In the same vein, we want to use a *SolutionsConfiguration* object, stored in `/etc/metalk8s/solutions.yaml`, to declare which *Solutions* are available to the cluster, from the bootstrap node.

Todo: Add specification in a future Reference guide

Here is how it could look:

```
apiVersion: metalk8s.scality.com/v1alpha1
kind: SolutionsConfiguration
solutions:
  - /solutions/storage_1.0.0.iso
  - /solutions/storage_latest.iso
  - /other_solutions/computing.iso
```

There would be no explicit information about what an archive contains. Instead, we want the archive itself to contain such information (more details in [Solution archive guidelines](#)), and to discover it at import time.

Note that Solutions will be **imported** based on this file contents, i.e. the images they contain will be made available in the registry and the UI will be deployed, however **deploying** the Operator and subsequent application(s) is left to the user, through manual operations or the Solution UI.

Note: Removing an archive path from the solutions list will effectively remove the Solution images and UI when the “import solutions” playbook is run.

25.1.3 Responsibilities of each party

This section intends to define the boundaries between MetalK8s and the Solutions to integrate with, in terms of “who is doing what?”.

Note: This is still a work in progress.

MetalK8s

MUST:

- Handle reading and mounting of the Solution ISO archive
- Provide tooling to deploy/upgrade a Solution’s CRDs and UI

MAY:

- Provide tooling to deploy/upgrade a Solution’s Operator
- Provide tooling to verify signatures in a Solution ISO
- Expose management of Solutions in its own UI

Solution

MUST:

- Comply with the standard archive structure defined by MetalK8s
- If providing a UI, expose management of its Operator instances
- Handle monitoring of its own services (both Operator and application, except the UI)

SHOULD:

- Use MetalK8s monitoring services (Prometheus and Grafana)

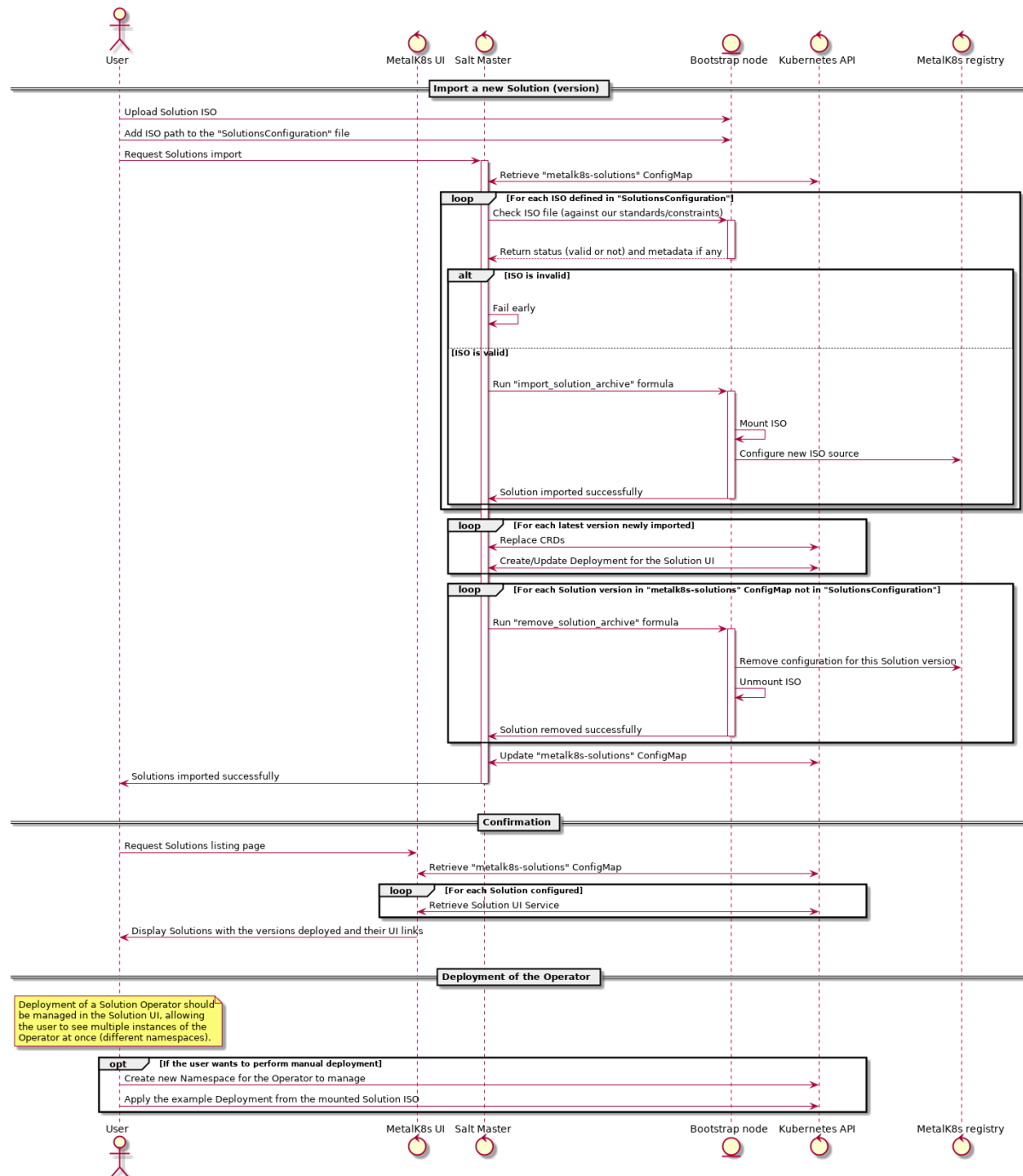
Note: Solutions can leverage the [Prometheus Operator](#) CRs for setting up the monitoring of their components. For more information, see [Monitoring](#) and [Solution Operator guidelines](#).

Todo: Define how Solutions can deploy Grafana dashboards.

25.1.4 Interaction diagrams

We include a detailed interaction sequence diagram for describing how MetalK8s will handle user input when deploying / upgrading Solutions.

Note: Open the image in a new tab to see it in full resolution.



Todo: A detailed diagram for Operator deployment would be useful (wait for #1060 to land). Also, add another diagram for specific operations in an upgrade scenario using two Namespaces, for staging/testing the new version.

25.2 Solution archive guidelines

To provide a predictable interface with packaged Solutions, MetalK8s expects a few criteria to be respected, described below.

25.2.1 Archive format

Solution archives must use the [ISO-9660:1988](#) format, including [Rock Ridge](#) and [Joliet](#) directory records. The character encoding must be [UTF-8](#). The conformance level is expected to be at most 3, meaning:

- Directory identifiers may not exceed 31 characters (bytes) in length
- File name + '.' + file name extension may not exceed 30 characters (bytes) in length
- Files are allowed to consist of multiple sections

The generated archive should specify a volume ID, set to {project_name} {version}.

Todo: Clarify whether Joliet/Rock Ridge records supersede the conformance level w.r.t. filename lengths

Here is an example invocation of the common Unix [mkisofs](#) tool to generate such archive:

```
mkisofs
  -output my_solution.iso
  -R # (or "-rock" if available)
  -J # (or "-joliet" if available)
  -joliet-long
  -l # (or "-full-iso9660-filenames" if available)
  -V 'MySolution 1.0.0' # (or "-volid" if available)
  -gid 0
  -uid 0
  -iso-level 3
  -input-charset utf-8
  -output-charset utf-8
  my_solution_root/
```

Todo: Consider if overriding the source files UID/GID to 0 is necessary

25.2.2 File hierarchy

Here is the file tree expected by MetalK8s to exist in each Solution archive:

```
.
├── images
│   └── some_image_name
│       └── 1.0.1
│           ├── <layer_digest>
│           ├── manifest.json
│           └── version
```

(continues on next page)

(continued from previous page)

```

├── registry-config.inc
├── operator
│   └── deploy
│       ├── crds
│       │   └── some_crd_name.yaml
│       ├── operator.yaml
│       ├── role.yaml
│       ├── role_binding.yaml
│       └── service_account.yaml
├── product.txt
├── ui
│   └── deployment.yaml

```

25.2.3 Product information

General product information about the packaged Solution must be stored in the `product.txt` file, stored at the archive root.

It must respect the following format (currently version 1, as specified by the `ARCHIVE_LAYOUT_VERSION` value):

```

NAME=Example
VERSION=1.0.0-dev
REQUIRE_METALK8S=">=2.0"
ARCHIVE_LAYOUT_VERSION=1

```

It is recommended for inspection purposes to include information related to the build-time conditions, such as the following (where command invocations should be statically replaced in the generated `product.txt`):

```

GIT=$(git describe --always --long --tags --dirty)
BUILD_TIMESTAMP=$(date +%Y-%m-%dT%H:%M:%SZ)

```

Note: If a Solution can require specific versions of MetalK8s on which to be deployed, requiring specific services (and their respective versions) to be shipped with MetalK8s (e.g. Prometheus/Grafana) is not yet feasible. It will probably be handled in the Operator declaration, maybe using a CR.

It is recommended for inspection purposes to include information related to the build-time conditions, such as the following (where command invocations should be statically replaced in the generated `product.txt`):

```

GIT=$(git describe --always --long --tags --dirty)
BUILD_TIMESTAMP=$(date +%Y-%m-%dT%H:%M:%SZ)

```

25.2.4 OCI images

MetalK8s exposes container images in the [OCI](#) format through a static read-only registry. This registry is built with [nginx](#), and relies on having a specific layout of image layers to then replicate the necessary parts of the Registry API that CRI clients (such as `containerd` or `cri-o`) rely on.

Using [skopeo](#), you can save images as a directory of layers:

```

$ mkdir images/my_image
$ # from your local Docker daemon
$ skopeo copy --format v2s2 --dest-compress docker-daemon:my_image:1.0.0 dir:images/my_image/1.0.0

```

(continues on next page)

(continued from previous page)

```
$ # from Docker Hub
$ skopeo copy --format v2s2 --dest-compress docker://docker.io/example/my_image:1.0.0 dir:images/my_
↪image/1.0.0
```

Your images directory should now resemble this:

```
images
├── my_image
│   └── 1.0.0
│       ├── 53071b97a88426d4db86d0e8436ac5c869124d2c414caf4c9e4a4e48769c7f37
│       ├── 64f5d945efcc0f39ab11b3cd4ba403cc9fefe1fa3613123ca016cf3708e8cafb
│       ├── manifest.json
│       └── version
```

Once all your images were stored this way, you can de-duplicate layers using hardlinks, using the tool [hardlink](#):

```
$ hardlink -c images
```

A detailed procedure for generating the expected layout is available at [NicolasT/static-container-registry](#). You can use the script provided there, or use the one vendored in this repository (located at `buildchain/buildchain/static-container-registry`) to generate the NGINX configuration to serve these image layers with the Docker Registry API. MetalK8s, when deploying the Solution, will include the `registry-config.inc` file provided at the root of the archive. In order to let MetalK8s control the mountpoint of the ISO, the configuration **must** be generated using the following options:

```
$ ./static-container-registry.py \
  --name-prefix '{{ repository }}' \
  --server-root '{{ registry_root }}' \
  /path/to/archive/images > /path/to/archive/registry-config.inc.j2
```

Each archive will be exposed as a single repository, where the name will be computed as `<NAME>-<VERSION>` from [Product information](#), and will be mounted at `/srv/scality/<NAME>-<VERSION>`.

Warning: Operators should not rely on this naming pattern for finding the images for their resources. Instead, the full repository prefix will be exposed to the Operator container as an environment variable when deployed with MetalK8s. See [Solution Operator guidelines](#) for more details.

The images names and tags will be inferred from the directory names chosen when using `skopeo copy`. Using `hardlink` is highly recommended if one wants to define alias tags for a single image.

MetalK8s also defines recommended standards for container images, described in [Container Images](#).

25.2.5 Operator

See *Solution Operator guidelines* for how the /operator directory should be populated.

25.2.6 Web UI

Todo: Create UI guidelines and reference here

25.3 Solution Operator guidelines

An Operator is a method of packaging, deploying and managing a Kubernetes application. A Kubernetes application is an application that is both deployed on Kubernetes and managed using the Kubernetes APIs and kubectl tooling.

—coreos.com/operators

MetalK8s *Solutions* are a concept mostly centered around the Operator pattern. While there is no explicit requirements except the ones described below (see *Requirements*), we recommend using the *Operator SDK* as it will embed best practices from the *Kubernetes* community. We also include some *Recommendations*.

25.3.1 Requirements

Files

All Operator-related files except for the container images (see *OCI images*) should be stored under /operator in the ISO archive. Those files should be organized as follows:

```
operator
├── deploy
│   ├── crds
│   │   └── some_crd_name.yaml
│   ├── operator.yaml
│   ├── role.yaml
│   ├── role_binding.yaml
│   └── service_account.yaml
```

Most of these files are generated when using the Operator SDK.

Todo: Specify each of them, include example (after #1060 is done). Remember to note specificities about OCI_REPOSITORY_PREFIX / namespaces. Think about using kustomize (or kubectl apply -k, though only available from K8s 1.14).

Monitoring

MetalK8s does not handle the monitoring of a Solution application, which means:

- the user, manually or through the Solution UI, should create Service and ServiceMonitor objects for each Operator instance
- Operators should create Service and ServiceMonitor objects for each deployed component they own

The [Prometheus Operator](#) deployed by MetalK8s has cluster-scoped permissions, and is able to read the aforementioned ServiceMonitor objects to set up monitoring of your application services.

25.3.2 Recommendations

Permissions

MetalK8s does not provide tools to deploy the Operator itself, so that users can have better control over which version runs where.

The best-practice encouraged here is to use namespace-scoped permissions for the Operator, instead of cluster-scoped.

This allows for better isolation between different application deployments from a single Solution, for instance when trying out a new version before affecting production machines, or when managing two independent application stacks.

Note: Future improvements to MetalK8s may include the addition of an “Operator for Operators”, such as the [Operator Lifecycle Manager](#).

25.4 Deploying And Experimenting

Given the solution ISO is correctly generated, a script utility has been added to manage Solutions. This script is located at the root of MetalK8s archive:

```
/srv/scality/metal8s-2.4.3/solutions.sh
```

25.4.1 Import a Solution

Importing a Solution will mount its ISO and expose its container images.

To import a Solution into MetalK8s cluster, use the import subcommand:

```
./solutions.sh import --archive </path/to/solution.iso>
```

The `--archive` option can be provided multiple times to import several Solutions ISOs at the same time:

```
./solutions.sh import --archive </path/to/solution1.iso> \  
--archive </path/to/solution2.iso>
```

25.4.2 Unimport a Solution

To unimport a Solution from MetalK8s cluster, use the unimport subcommand:

Warning: Images of a Solution will no longer be available after an archive removal

```
./solutions.sh unimport --archive </path/to/solution.iso>
```

25.4.3 Activate a Solution

Activating a Solution version will deploy its CRDs.

To activate a Solution in MetalK8s cluster, use the activate subcommand:

```
./solutions.sh activate --name <solution-name> --version <solution-version>
```

25.4.4 Deactivate a Solution

To deactivate a Solution from MetalK8s cluster, use the deactivate subcommand:

```
./solutions.sh deactivate --name <solution-name>
```

25.4.5 Create an Environment

To create a Solution Environment, use the create-env subcommand:

```
./solutions.sh create-env --name <environment-name>
```

By default, it will create a Namespace named after the <environment-name>, but it can be changed, using the --namespace option:

```
./solutions.sh create-env --name <environment-name> \
  --namespace <namespace-name>
```

It's also possible to use the previous command to create multiple Namespaces (one at a time) in this Environment, allowing Solutions to run in different Namespaces.

25.4.6 Delete an Environment

To delete an Environment, use the delete-env subcommand:

Warning: This will destroy everything in the said Environment, with no way back

```
./solutions.sh delete-env --name <environment-name>
```

In case of multiple Namespaces inside an Environment, it's also possible to only delete a single Namespace, using:

```
./solutions.sh delete-env --name <environment-name> \
  --namespace <namespace-name>
```

25.4.7 Add a Solution in an Environment

Adding a Solution will deploy its UI and Operator resources in the Environment.

To add a Solution in an Environment, use the `add-solution` subcommand:

```
./solutions.sh add-solution --name <environment-name> \  
  --solution <solution-name> --version <solution-version>
```

In case of non-default Namespace (not corresponding to `<environment-name>`) or multiple Namespaces in an Environment, Namespace in which the Solution will be added must be precised, using the `--namespace` option:

```
./solutions.sh add-solution --name <environment-name> \  
  --solution <solution-name> --version <solution-version> \  
  --namespace <namespace-name>
```

25.4.8 Delete a Solution from an Environment

To delete a Solution from an Environment, use the `delete-solution` subcommand:

```
./solutions.sh delete-solution --name <environment-name> \  
  --solution <solution-name>
```

25.4.9 Upgrade/Downgrade a Solution

Before starting, the destination version must have been imported.

Patch the Environment ConfigMap, with the destination version:

```
kubect1 patch cm metalk8s-environment --namespace <namespace-name> \  
  --patch '{"data": {"<solution-name>": "<solution-version-dest>"}}'
```

Apply the change with Salt:

```
salt_container=$(  
  crictl ps -q \  
  --label io.kubernetes.pod.namespace=kube-system \  
  --label io.kubernetes.container.name=salt-master \  
  --state Running  
)  
crictl exec -i "$salt_container" salt-run state.orchestrate \  
  metalk8s.orchestrate.solutions.prepare-environment \  
  pillar="{ 'orchestrate': { 'env_name': '<environment-name>' } }"
```

Part IV

Glossary

Alertmanager The Alertmanager is a service for handling alerts sent by client applications, such as [Prometheus](#).

See also the official Prometheus documentation for [Alertmanager](#).

API Server

kube-apiserver The Kubernetes API Server validates and configures data for the Kubernetes objects that make up a cluster, such as [Nodes](#) or [Pods](#).

See also the official Kubernetes documentation for [kube-apiserver](#).

Bootstrap

Bootstrap node The Bootstrap node is the first machine on which MetalK8s is installed, and from where the cluster will be deployed to other machines. It also serves as the entrypoint for upgrades of the cluster.

ConfigMap A ConfigMap is a Kubernetes object that allows one to store general configuration information such as environment variables in a key-value pair format. ConfigMaps can only be applied to namespaces and once created, they can be updated automatically without the need of restarting containers that depend on it.

See also the official Kubernetes documentation for [ConfigMap](#).

Controller Manager

kube-controller-manager The Kubernetes controller manager embeds the core control loops shipped with Kubernetes, which role is to watch the shared state from [API Server](#) and make changes to move the current state towards the desired state.

See also the official Kubernetes documentation for [kube-controller-manager](#).

etcd etcd is a distributed data store, which is used in particular for the persistent storage of [API Server](#).

For more information, see [etcd.io](#).

Kubeconfig A configuration file for [kubectl](#), which includes authentication through embedded certificates.

See also the official Kubernetes documentation for [kubeconfig](#).

Kubelet The kubelet is the primary “node agent” that runs on each cluster node.

See also the official Kubernetes documentation for [kubelet](#).

Namespace A Namespace is a Kubernetes abstraction to support multiple virtual clusters backed by the same physical cluster, providing a scope for resource names.

See also the official Kubernetes documentation for [namespaces](#).

Node A Node is a Kubernetes worker machine - either virtual or physical. A Node contains the services required to run [Pods](#).

See also the official Kubernetes documentation for [Nodes](#).

Node manifest The YAML file describing a [Node](#).

See also the official Kubernetes documentation for [Nodes management](#).

Pod A Pod is a group of one or more containers sharing storage and network resources, with a specification of how to run these containers.

See also the official Kubernetes documentation for [Pods](#).

Prometheus Prometheus serves as a time-series database, and is used in MetalK8s as the storage for all metrics exported by applications, whether being provided by the cluster or installed afterwards.

For more details, see [prometheus.io](#).

SaltAPI SaltAPI is an HTTP service for exposing operations to perform with a [Salt Master](#). The version deployed by MetalK8s is configured to use the cluster authentication/authorization services.

See also the official SaltStack documentation for [SaltAPI](#).

Salt Master The Salt Master is a daemon responsible for orchestrating infrastructure changes by managing a set of [Salt Minions](#).

See also the official SaltStack documentation for [Salt Master](#).

Salt Minion The Salt Minion is an agent responsible for operating changes on a system. It runs on all MetalK8s nodes.

See also the official SaltStack documentation for [Salt Minion](#).

Scheduler

kube-scheduler The Kubernetes scheduler is responsible for assigning [Pods](#) to specific [Nodes](#) using a complex set of constraints and requirements.

See also the official Kubernetes documentation for [kube-scheduler](#).

Service A Kubernetes Service is an abstract way to expose an application running on a set of [Pods](#) as a network service.

See also the official Kubernetes documentation for [Services](#).

Taint Taints are a system for Kubernetes to mark [Nodes](#) as reserved for a specific use-case. They are used in conjunction with [tolerations](#).

See also the official Kubernetes documentation for [taints and tolerations](#).

Toleration Tolerations allow to mark [Pods](#) as schedulable for all [Nodes](#) matching some *filter*, described with [taints](#).

See also the official Kubernetes documentation for [taints and tolerations](#).

kubect1 `kubect1` is a CLI interface for interacting with a Kubernetes cluster.

See also the official Kubernetes documentation for [kubect1](#).

A

Alertmanager, [139](#)

API Server, [139](#)

B

Bootstrap, [139](#)

Bootstrap node, [139](#)

C

ConfigMap, [139](#)

Controller Manager, [139](#)

E

etcd, [139](#)

K

kube-apiserver, [139](#)

kube-controller-manager, [139](#)

kube-scheduler, [140](#)

Kubeconfig, [139](#)

kubect1, [140](#)

Kubelet, [139](#)

N

Namespace, [139](#)

Node, [139](#)

Node manifest, [139](#)

P

Pod, [139](#)

Prometheus, [139](#)

S

Salt Master, [140](#)

Salt Minion, [140](#)

SaltAPI, [140](#)

Scheduler, [140](#)

Service, [140](#)

T

Taint, [140](#)

Toleration, [140](#)