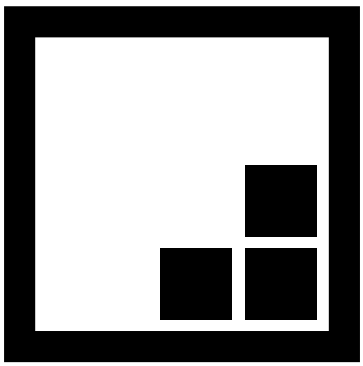


METALK8S

MetalK8s Documentation

Release 2.4.0-beta2

Scality

Oct 12, 2019

CONTENTS:

I Quickstart Guide 1

1 Introduction 5

2 Setup of the environment 9

3 Deployment of the Bootstrap node 11

4 Cluster expansion 15

5 Accessing cluster services 21

II Installation Guide 25

6 Sizing recommendations 27

III Operational Guide 29

7 Bootstrap Node Backup and Restoration Procedure 33

8 ISO Preparation 35

9 Upgrade Guide 37

10 Downgrade Guide 39

11 Changing the hostname of a MetalK8s node 41

12 Volume Management 43

IV Developer Guide 49

13 Architecture Documents 51

14 Design Documents 61

15 How to build MetalK8s 69

16 How to run components locally 73

17 Development Best Practices 77

18 Integrating with MetalK8s 81

V Glossary 89

Index 93

Part I

Quickstart Guide

This guide describes how to set up a MetalK8s cluster for experimentation. For production installations, refer to the *Installation Guide*. It offers general requirements and describes sizing, configuration, and deployment. It also explains major concepts central to MetalK8s architecture, and will show how to access various services after completing the setup.

CHAPTER

ONE

INTRODUCTION

1.1 Concepts

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

1.1.1 Nodes

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

1.1.2 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).

1.1.3 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/node This role marks a workload-plane node. It is included implicitly by all other roles.

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 will be used in conjunction with the `master` and `etcd` roles for bootstrapping the control-plane.

1.1.4 Node taints

Taints are complementary to roles. When a taint, or a set of taints, are 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.

1.1.5 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:

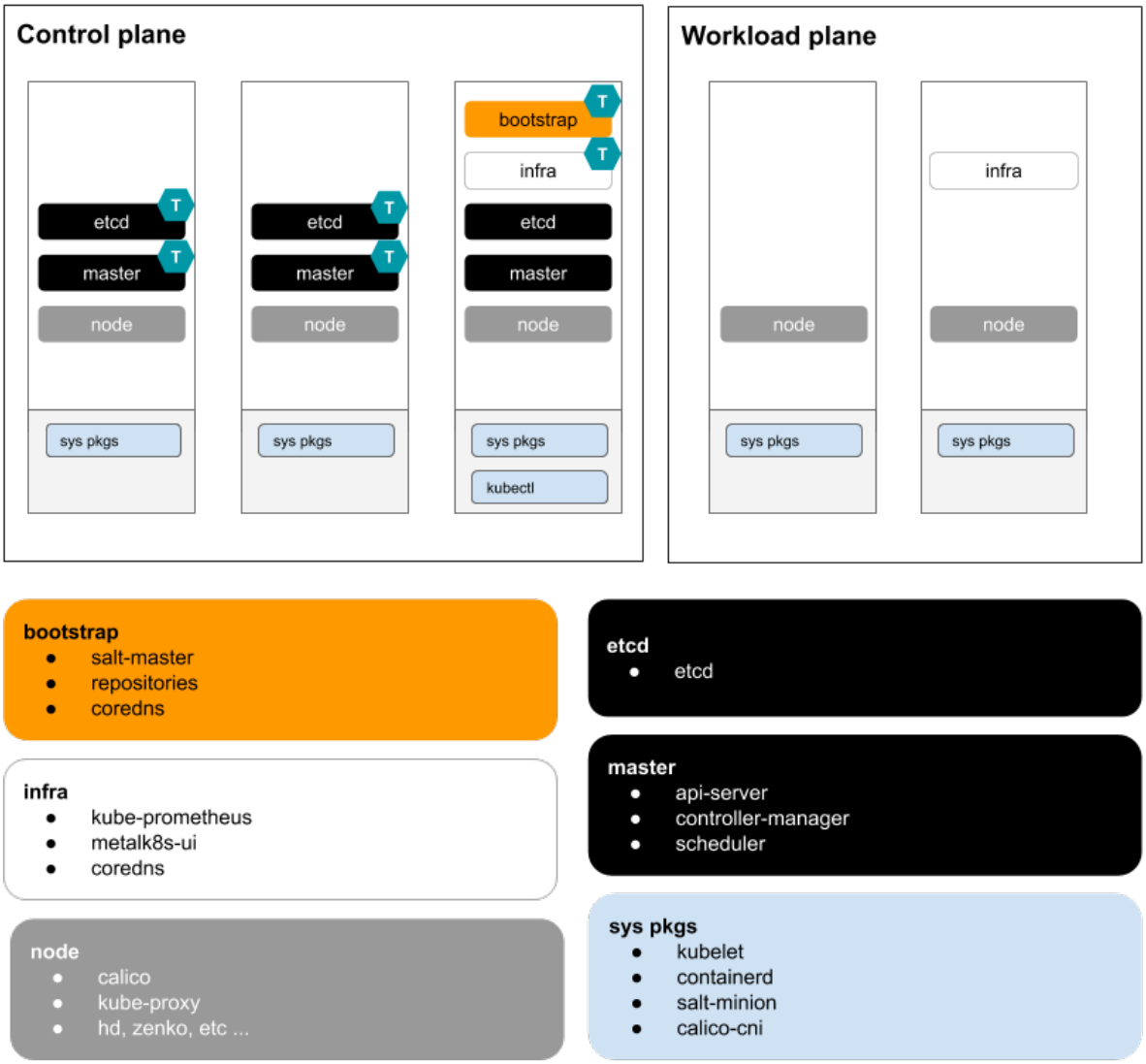
- A network for *Pods*, defaulting to `10.233.0.0/16`
- A network for *Services*, defaulting to `10.96.0.0/12`

In case of conflicts with the existing infrastructure, make sure to choose other ranges during the *Bootstrap configuration*.

1.2 Installation plan

In this guide, the depicted installation procedure is for a medium sized cluster, using three control-plane nodes and two worker nodes. Refer to the *Installation Guide* for extensive explanations of possible cluster architectures.

Note: This image depicts the architecture deployed with this Quickstart guide.



Todo:

- describe architecture schema, include legend
- improve architecture explanation and presentation

The installation process can be broken down into the following steps:

1. *Setup* of the environment (with requirements and example OpenStack deployment)
2. *Deployment* of the *Bootstrap node*
3. *Expansion* of the cluster from the Bootstrap node

Todo: Include a link to example Solution deployment?

CHAPTER

TWO

SETUP OF THE ENVIRONMENT

2.1 General requirements

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.

For this quickstart, we will need 5 machines (or 3, if running workload applications on your control-plane nodes).

2.1.1 Sizing

Each machine should have at least 2 CPU cores, 4 GB of RAM, and a root partition larger than 40 GB.

For sizing recommendations depending on sample use cases, see the *Installation guide*.

2.1.2 Proxies

For nodes operating behind a proxy, add the following lines to each cluster member's `/etc/environment` file:

```
http_proxy=http://user;pass@<HTTP proxy IP address>:<port>
https_proxy=http://user;pass@<HTTPS proxy IP address>:<port>
no_proxy=localhost,127.0.0.1,<local IP of each node>
```

2.1.3 SSH provisioning

Each machine should be accessible through SSH from your 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.

2.1.4 Network provisioning

Each machine needs to 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.

In order to reach the cluster-provided UIs from your host, the host needs to be able to connect to workload-plane IPs of the machines.

2.2 Example OpenStack deployment

Todo: Extract the Terraform tooling used in CI for ease of use.

CHAPTER

THREE

DEPLOYMENT OF THE BOOTSTRAP NODE

3.1 Preparation

3.1.1 MetalK8s ISO

On your bootstrap node, download the MetalK8s ISO file. Mount this ISO file at the specific following path:

```
root@bootstrap $ mkdir -p /srv/scality/metalk8s-2.4.0-beta2
root@bootstrap $ mount <path-to-iso> /srv/scality/metalk8s-2.4.0-beta2
```

3.2 Configuration

1. Create the MetalK8s configuration directory.

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

2. Create the `/etc/metalk8s/bootstrap.yaml` file. Change the networks, IP address, and hostname to conform to your infrastructure.

   ```
   apiVersion: metalk8s.scality.com/v1alpha2
   kind: BootstrapConfiguration
   networks:
     controlPlane: <CIDR-notation>
     workloadPlane: <CIDR-notation>
   ca:
     minion: <hostname-of-the-bootstrap-node>
   apiServer:
     host: <IP-of-the-bootstrap-node>
   archives:
     - <path-to-metalk8s-iso>
   ```

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.

Todo:

- Explain the role of this config file and its values
- Add a note about setting HA for `apiServer`

3.2.1 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 occurences 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). First, retrieve the public key from the Bootstrap node.

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

Then, 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>
```

3.3 Installation

3.3.1 Run the install

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

```
root@bootstrap $ /srv/scality/metalk8s-2.4.0-beta2/bootstrap.sh
```

3.3.2 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. Templates for the required volumes are available in `examples/prometheus-sparse.yaml`. Note, however, these templates use the *sparseLoopDevice Volume* type, which is not suitable for production installations. Refer to *Volume Management* for more information on how to provision persistent storage.

Note: When deploying using Vagrant, persistent volumes for Prometheus and AlertManager are already provisioned.

3.3.3 Validate the install

Check if all *Pods* on the Bootstrap node are in the `Running` state.

Note: 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       v1.11.7

root@bootstrap $ kubectl get pods --all-namespaces -o wide --kubeconfig /etc/kubernetes/admin.conf
NAMESPACE             NAME                                          READY     STATUS    RESTARTS␣
↪  AGE       IP              NODE        NOMINATED NODE
kube-system           calico-kube-controllers-b7bc4449f-6rh2q       1/1       Running   0       ␣
↪  4m        10.233.132.65   bootstrap   <none>
kube-system           calico-node-r2qxs                             1/1       Running   0       ␣
↪  4m        172.21.254.12   bootstrap   <none>
kube-system           coredns-7475f8d796-8h4lt                      1/1       Running   0       ␣
↪  4m        10.233.132.67   bootstrap   <none>
kube-system           coredns-7475f8d796-m5zz9                      1/1       Running   0       ␣
↪  4m        10.233.132.66   bootstrap   <none>
kube-system           etcd-bootstrap                                1/1       Running   0       ␣
↪  4m        172.21.254.12   bootstrap   <none>
kube-system           kube-apiserver-bootstrap                      2/2       Running   0       ␣
↪  4m        172.21.254.12   bootstrap   <none>
kube-system           kube-controller-manager-bootstrap             1/1       Running   0       ␣
↪  4m        172.21.254.12   bootstrap   <none>
kube-system           kube-proxy-vb74b                              1/1       Running   0       ␣
↪  4m        172.21.254.12   bootstrap   <none>
kube-system           kube-scheduler-bootstrap                      1/1       Running   0       ␣
↪  4m        172.21.254.12   bootstrap   <none>
kube-system           repositories-bootstrap                        1/1       Running   0       ␣
↪  4m        172.21.254.12   bootstrap   <none>
kube-system           salt-master-bootstrap                         2/2       Running   0       ␣
↪  4m        172.21.254.12   bootstrap   <none>
metalk8s-ingress      nginx-ingress-controller-46lxd                1/1       Running   0       ␣
↪  4m        10.233.132.73   bootstrap   <none>
metalk8s-ingress      nginx-ingress-default-backend-5449d5b699-8bkbr 1/1       Running   0       ␣
↪  4m        10.233.132.74   bootstrap   <none>
metalk8s-monitoring   alertmanager-main-0                           2/2       Running   0       ␣
↪  4m        10.233.132.70   bootstrap   <none>
metalk8s-monitoring   alertmanager-main-1                           2/2       Running   0       ␣
↪  3m        10.233.132.76   bootstrap   <none>
metalk8s-monitoring   alertmanager-main-2                           2/2       Running   0       ␣
↪  3m        10.233.132.77   bootstrap   <none>
metalk8s-monitoring   grafana-5cb4945b7b-ltdrz                      1/1       Running   0       ␣
↪  4m        10.233.132.71   bootstrap   <none>
metalk8s-monitoring   kube-state-metrics-588d699b56-d6crn            4/4       Running   0       ␣
↪  3m        10.233.132.75   bootstrap   <none>
metalk8s-monitoring   node-exporter-4jdgv                           2/2       Running   0       ␣
↪  4m        172.21.254.12   bootstrap   <none>
metalk8s-monitoring   prometheus-k8s-0                              3/3       Running   1       ␣
↪  4m        10.233.132.72   bootstrap   <none>
metalk8s-monitoring   prometheus-k8s-1                              3/3       Running   1       ␣
↪  3m        10.233.132.78   bootstrap   <none>
metalk8s-monitoring   prometheus-operator-64477d4bff-xxjw2          1/1       Running   0       ␣
↪  4m        10.233.132.68   bootstrap   <none>
```

Check that you can access the MetalK8s GUI, following *this procedure*.

3.3.4 Troubleshooting

Todo:

- Mention `/var/log/metalk8s-bootstrap.log` and the command-line options for verbosity.
- Add Salt master/minion logs, and explain how to run a specific state from the Salt master.
- Then refer to a troubleshooting section in the installation guide.

CHAPTER

FOUR

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 entrypoint. This operation can be done either through *the MetalK8s GUI* or *the command-line*.

4.1 Defining an architecture

See the schema defined in *the introduction*.

The Bootstrap being already deployed, the deployment of other Nodes will need to happen four times, twice for control-plane Nodes (bringing up the control-plane to a total of three members), and twice for workload-plane Nodes.

Todo:

- explain architecture: 3 control-plane + etcd, 2 workers (one being dedicated for infra)
- remind roles and taints from intro

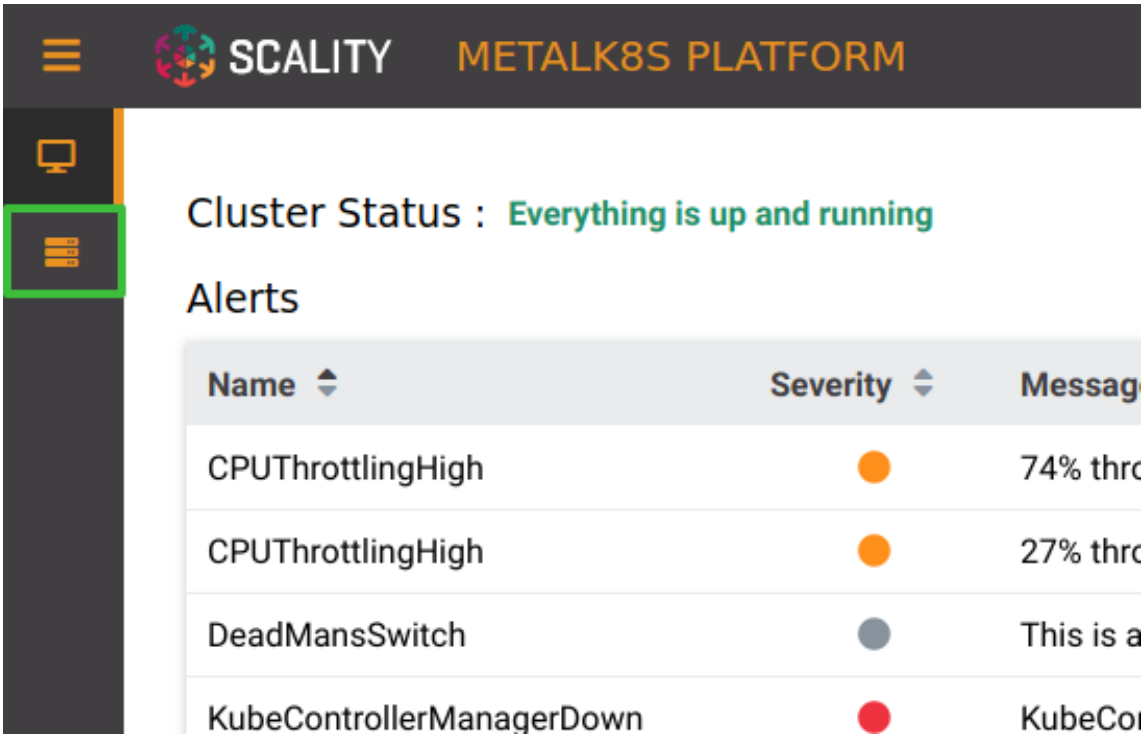
4.2 Adding a node with the MetalK8s GUI

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

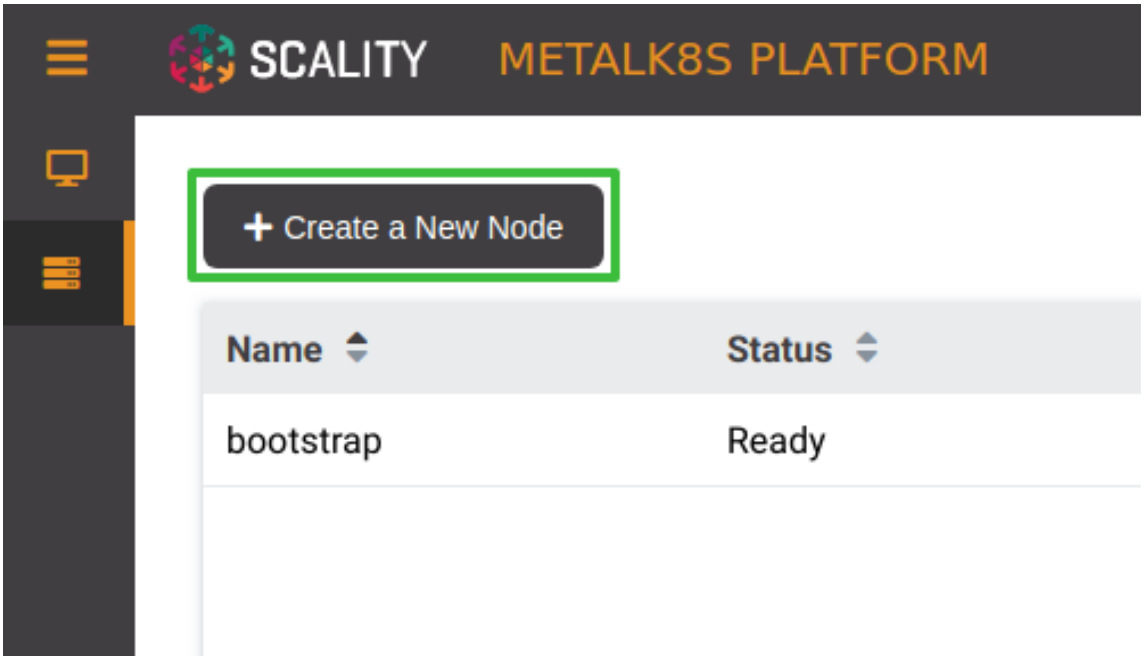
4.2.1 Creating a Node object

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:



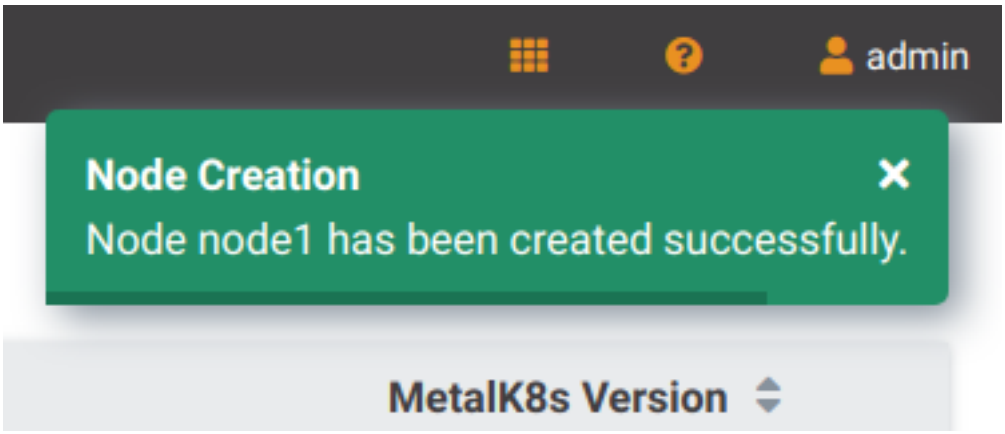
2. From the Node list (the Bootstrap node should be visible there), click the button labeled “Create a New Node”:



3. 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
   - **MetalK8s Version**: use “2.4.0-beta2”
   - **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**: check this box if the new Node should receive workload applications
- **Roles/Control Plane**: check this box if the new Node should run control-plane services
- **Roles/Infra**: check this box if the new Node should run infra services

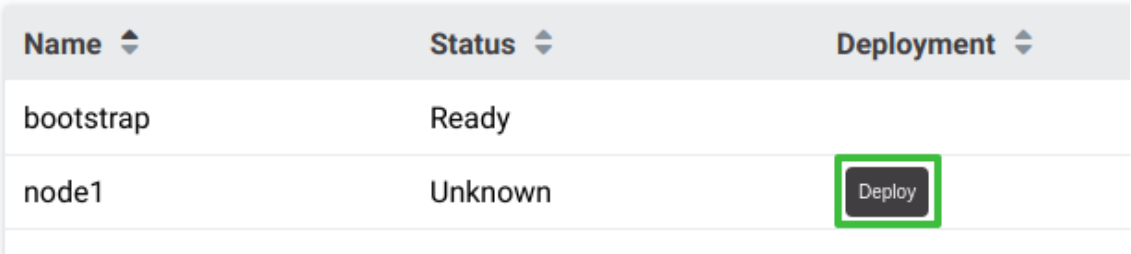
4. Click "Create". You will be redirected to the Node list page, and will be shown a notification to confirm the Node creation:



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, any yet-to-be-deployed Node will have a "Deploy" button. Click it to begin the deployment:



2. Once clicked, the button will change 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

3. When complete, click on "Back to nodes list". The new Node should have a `Ready` status.

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: '2.4.0-beta2'
    <role labels>
spec:
  taints: <taints>
```

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

A node exclusively in the control-plane with `etcd` storage will have:

```
[...]
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*) will use:

```
[...]
metadata:
  [...]
```

(continues on next page)

(continued from previous page)

```
  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.

4.3.2 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,master   12d       v1.11.7
<node-name>           Unknown   <expected node roles>         29s
```

4.3.3 Deploying the node

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

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
```

Start the node deployment.

```
root@salt-master-bootstrap $ salt-run state.orchestrate metalk8s.orchestrate.deploy_node \
                             saltenv=metalk8s-2.4.0-beta2 \
                             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
```

4.3.4 Troubleshooting

Todo:

- explain orchestrate output and how to find errors

- point to log files

4.4 Checking the 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)

CHAPTER

FIVE

ACCESSING CLUSTER SERVICES

5.1 MetalK8s GUI

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

5.1.1 Gather required information

1. Get the workload plane IP of the bootstrap node.

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

2. Retrieve the active `NodePort` number for the UI (here `30923`):

```
root@boostrap $ kubectl --kubeconfig=/etc/kubernetes/admin.conf get svc metalk8s-ui -n␣
↪metalk8s-ui

NAME                  TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
metalk8s-ui           NodePort    10.104.61.208   <none>        80:30923/TCP     3h
```

5.1.2 Use MetalK8s UI

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

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

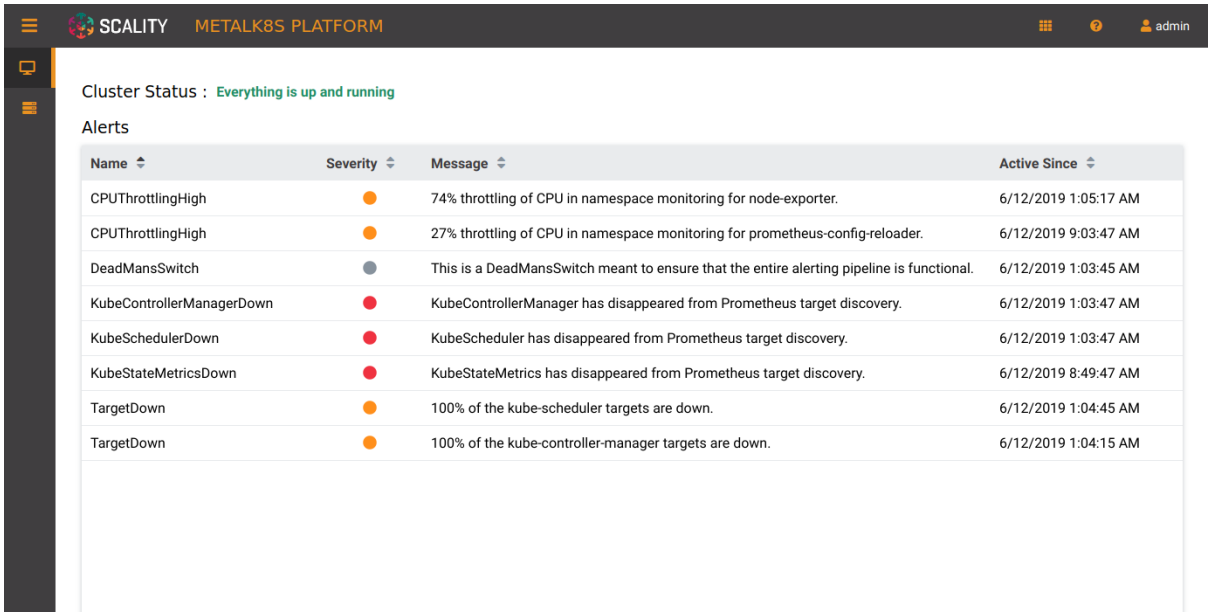


In the bottom left corner of the page, click the link `Accept SSL Certificate for Kubernetes`. In the new tab, click the button `Advanced...`, then select `Accept the risk and continue`.

Follow the same steps for the second link, `Accept SSL Certificate for Salt`.

Go back to the first tab, then 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*

5.2 Grafana

You will first need the latest `kubectl` version installed on your host.

To authenticate with the cluster, retrieve the admin kubeconfig on your host:

```
user@your-host $ scp root@bootstrap:/etc/kubernetes/admin.conf ./admin.conf
```

Forward the port used by Grafana:

```
user@your-host $ kubectl -n metalk8s-monitoring port-forward svc/prometheus-operator-grafana 3000:80
```

Then open your web browser and navigate to `http://localhost:3000`

5.3 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 (we assume 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

Installation Guide

CHAPTER

SIX

SIZING RECOMMENDATIONS

Todo: Evaluate requirements for various architectures

Part III

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 Guide* if you do not have a working MetalK8s setup.

CHAPTER

SEVEN

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: You cannot use the restore script if you do not have High Availability apiserver because some information required to reconfigure the others nodes are stored in the apiserver.

Warning: In case of a 3-node etcd cluster (2 nodes + unreachable old bootstrap node) you need to remove the old bootstrap node from the etcd cluster before running the restore script.

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>
```

Note: Replace *<backup_archive>* with the path to the backup archive you want to use.

CHAPTER

EIGHT

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>
```

CHAPTER

NINE

UPGRADE GUIDE

This section describes a reliable upgrade path for **MetalK8s** including all the components that make up the stack.

9.1 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.

9.2 Upgrade Pre-requisites

Prior to beginning the upgrade steps listed below, make sure to complete the pre-requisites listed in *ISO Preparation*.

9.3 Upgrade Steps

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

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

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

CHAPTER

TEN

DOWNGRADE GUIDE

This section describes the logical steps for downgrading **MetalK8s**.

To downgrade your cluster you need to run the utility script shipped with the current installation providing it with the destination version:

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

CHAPTER

ELEVEN

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
Oerating 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: '2.4.0-beta2'
    <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.
↪0-beta2   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
```

CHAPTER

TWELVE

VOLUME MANAGEMENT

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

12.1 StorageClass Creation

MetalK8s uses **StorageClass** objects to describe how **Volumes** are formatted and mounted. This section hightlights 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
   ```

12.2 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*

12.2.1 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.metalk8s.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   metalk8s-demo-storageclass
   ```

12.2.2 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.metalk8s.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
```

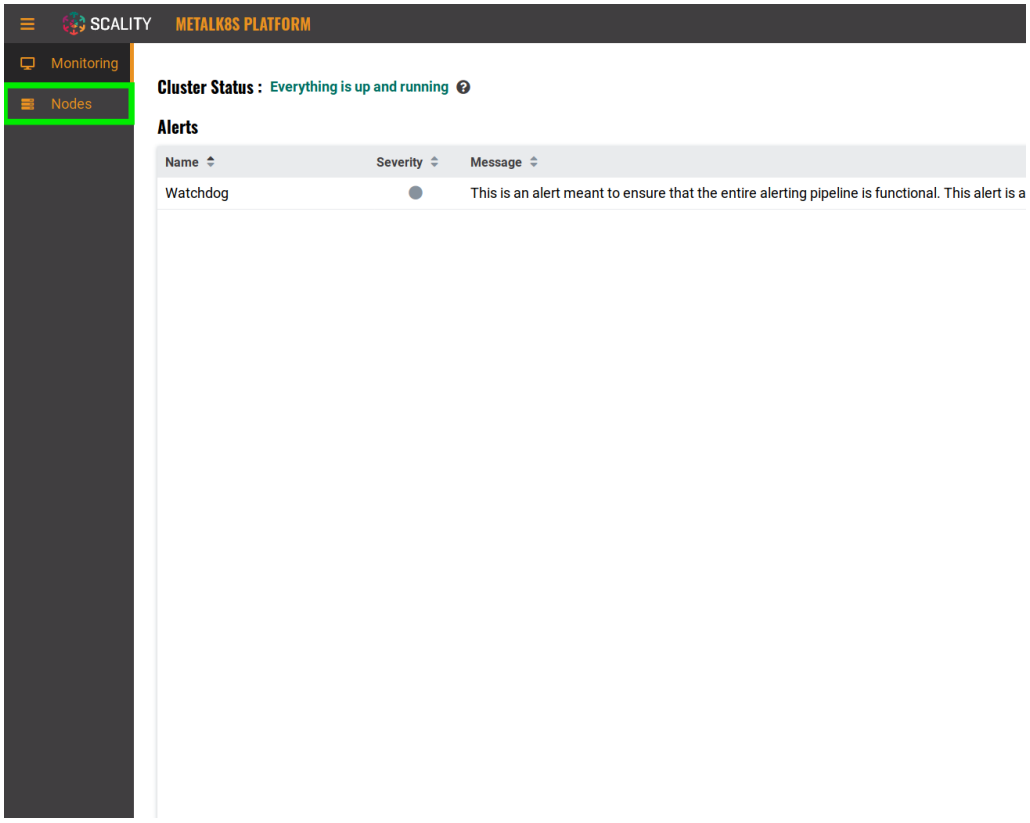
12.3 Volume Management using the UI

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*

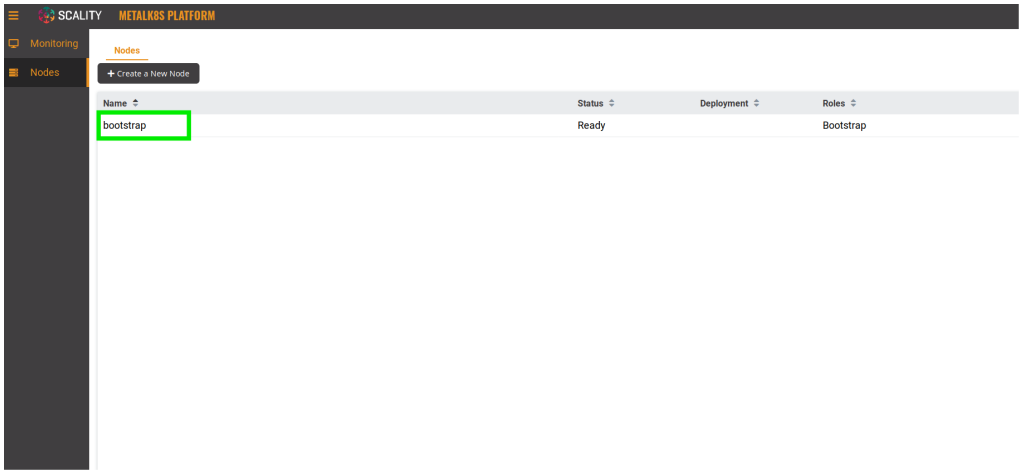
12.3.1 Volume Creation

To access the UI, refer to *this procedure*

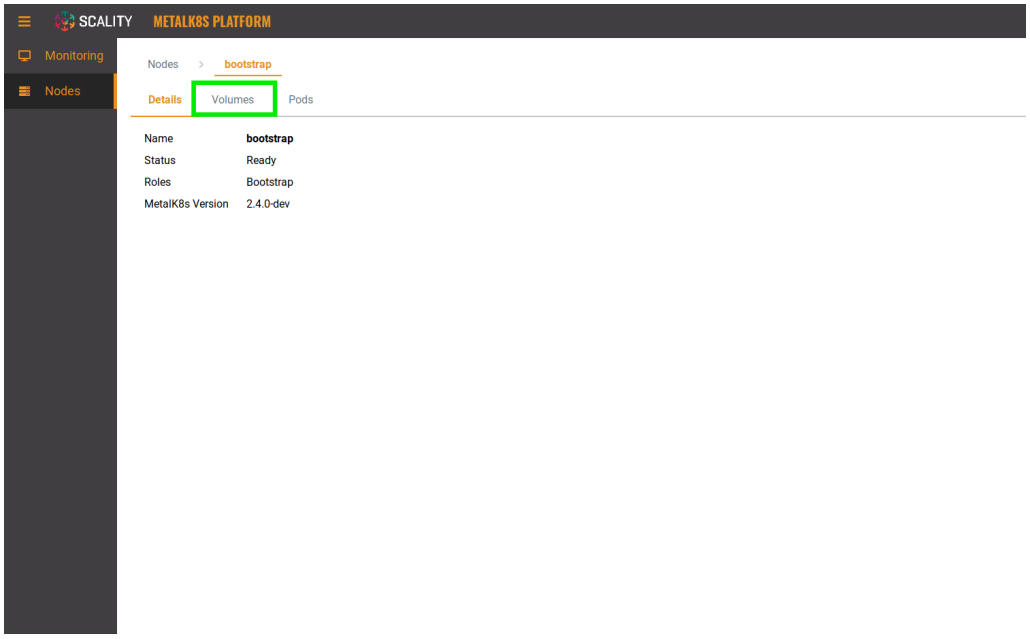
1. Navigate to the **Nodes** list page, by clicking the button in the sidebar:



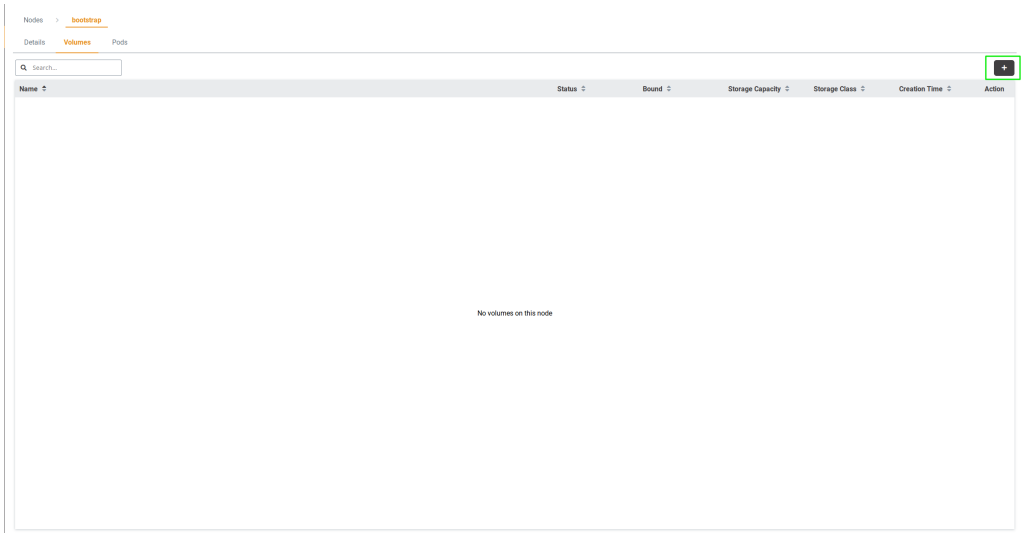
2. From the Node list, select the node you would like to create a volume on



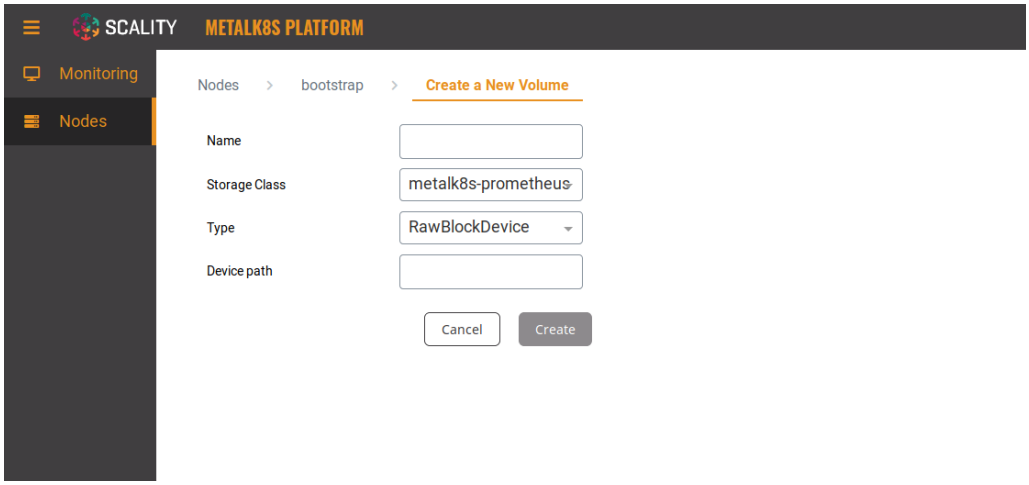
3. Navigate to the **Volumes** tab



4. Click the + button to create a volume

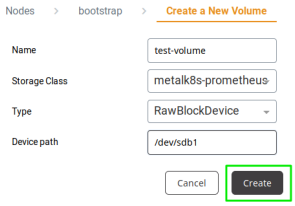


5. Fill out the respective fields

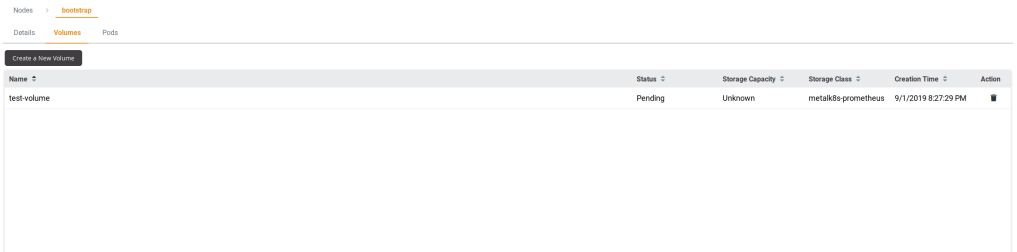


- **Name**: Denotes the volume name.
- **Storage Class**: Refer to the storage class creation page listed here: *StorageClass Creation*
- **Type**: Metalk8s currently only supports **RawBlockDevice** and **SparseLoopDevice**.
- **Device path**: Refers to the path of an existing storage device.

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

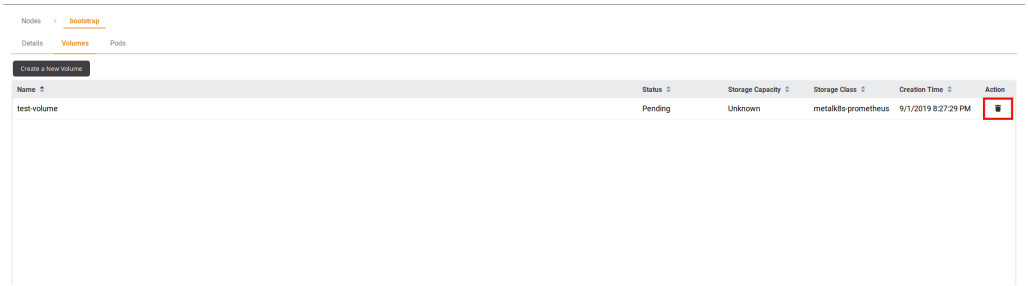


7. You should have a new volume listed in the **Volume list**

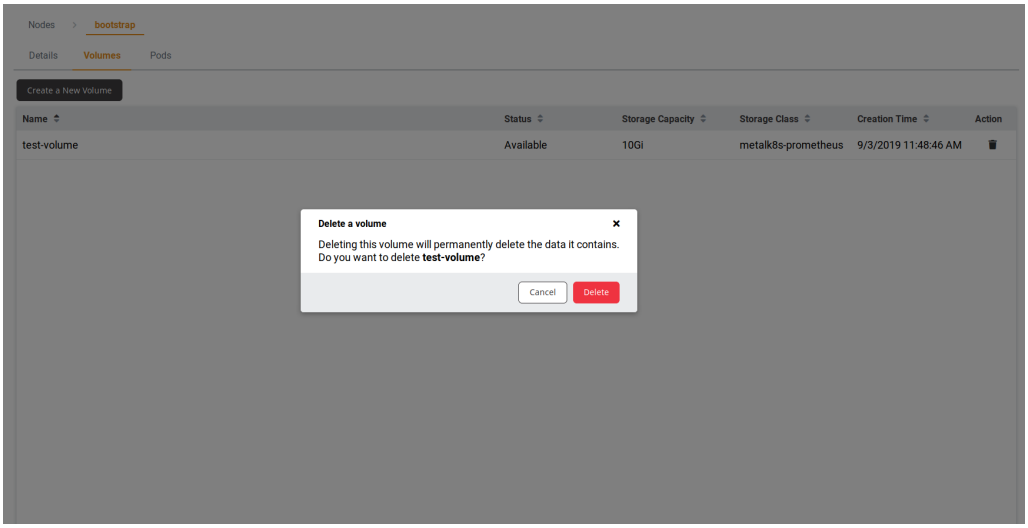


12.3.2 Volume Deletion

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



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



Part IV

Developer Guide

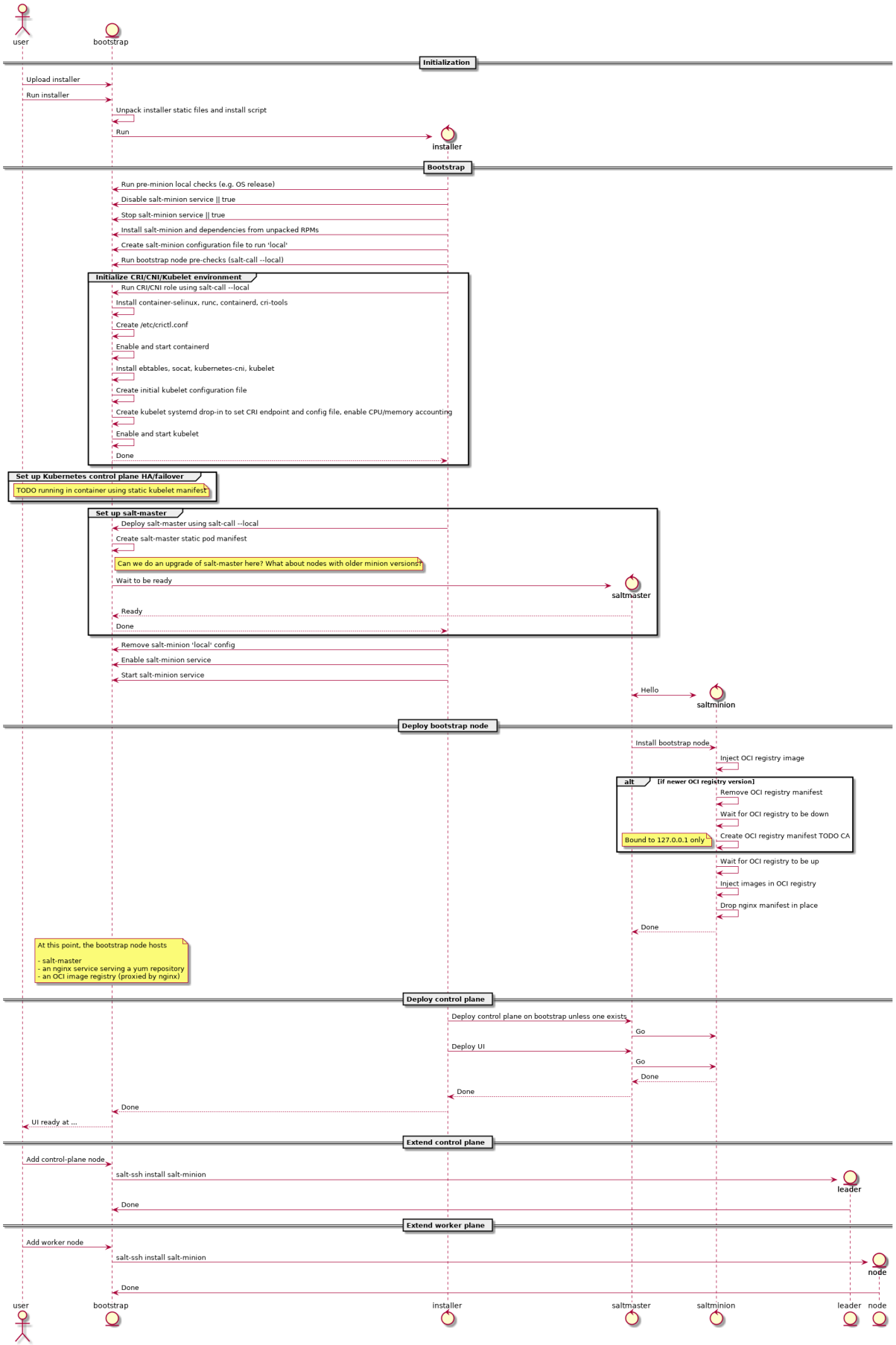
CHAPTER

THIRTEEN

ARCHITECTURE DOCUMENTS

13.1 Deployment

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


user
bootstrap
Initialization
Upload installer
Run installer
Unpack installer static files and install script
Run
installer
Bootstrap
Run pre-minion local checks (e.g. OS release)
Disable salt-minion service || true
Stop salt-minion service || true
Install salt-minion and dependencies from unpacked RPMs
Create salt-minion configuration file to run 'local'
Run bootstrap node pre-checks (salt-call --local)
Initialize CRI/CNI/Kubelet environment
Run CRI/CNI role using salt-call --local
Install container-selinux, runc, containerd, cri-tools
Create /etc/crictl.conf
Enable and start containerd
Install ebtables, socat, kubernetes-cni, kubelet
Create initial kubelet configuration file
Create kubelet systemd drop-in to set CRI endpoint and config file, enable CPU/memory accounting
Enable and start kubelet
Done
Set up Kubernetes control plane HA/failover
TODO running in container using static kubelet manifest
Set up salt-master
Deploy salt-master using salt-call --local
Create salt-master static pod manifest
Can we do an upgrade of salt-master here? What about nodes with older minion versions?
Wait to be ready
saltmaster
Ready
Done
Remove salt-minion 'local' config
Enable salt-minion service
Start salt-minion service
Hello
saltminion
Deploy bootstrap node
Install bootstrap node
Inject OCI registry image
alt
[if newer OCI registry version]
Remove OCI registry manifest
Wait for OCI registry to be down
Create OCI registry manifest TODO CA
Bound to 127.0.0.1 only
Wait for OCI registry to be up
Inject images in OCI registry
Drop nginx manifest in place
Done
At this point, the bootstrap node hosts
- salt-master
- an nginx service serving a yum repository
- an OCI image registry (proxied by nginx)
Deploy control plane
Deploy control plane on bootstrap unless one exists
Go
Deploy UI
Go
Done
Done
Done
UI ready at ...
Extend control plane
Add control-plane node
salt-ssh install salt-minion
leader
Done
Extend worker plane
Add worker node
salt-ssh install salt-minion
node
Done
user
bootstrap
installer
saltmaster
saltminion
leader
node

13.1.1 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`

- VIP for the `kube-apiserver` and `keepalived` toggle

  Used as the address of `kube-apiserver` where required. This can either be a VIP managed by custom load-balancing/high-availability infrastructure, in which case the `keepalived` toggle must be off, or one which our platform will manage using `keepalived`.

  If `keepalived` is enabled, this VIP must sit in a control plane CIDR shared by all control plane nodes.

  Note: we run `keepalived` in unicast mode, which is an extension of classic VRRP, but removes the need for multicast support on the network.

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:

- VRRP if `keepalived` is enabled on control-plane nodes
- 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

13.2 Monitoring

This document describes the monitoring features included in MetalK8s.

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

13.3 Requirements

13.3.1 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.

13.3.2 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.

13.3.3 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.

13.3.4 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

13.3.5 Storage

TODO

13.3.6 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).

CHAPTER

FOURTEEN

DESIGN DOCUMENTS

14.1 Volume Management v1.0

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

14.1.1 Absract

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.

14.1.2 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 (unformated) as **Volume**
- use an **Admission Controller** for semantic validation
- auto-discovery of the disks
- batch provisioning from the Platform UI

14.1.3 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 occured).

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.

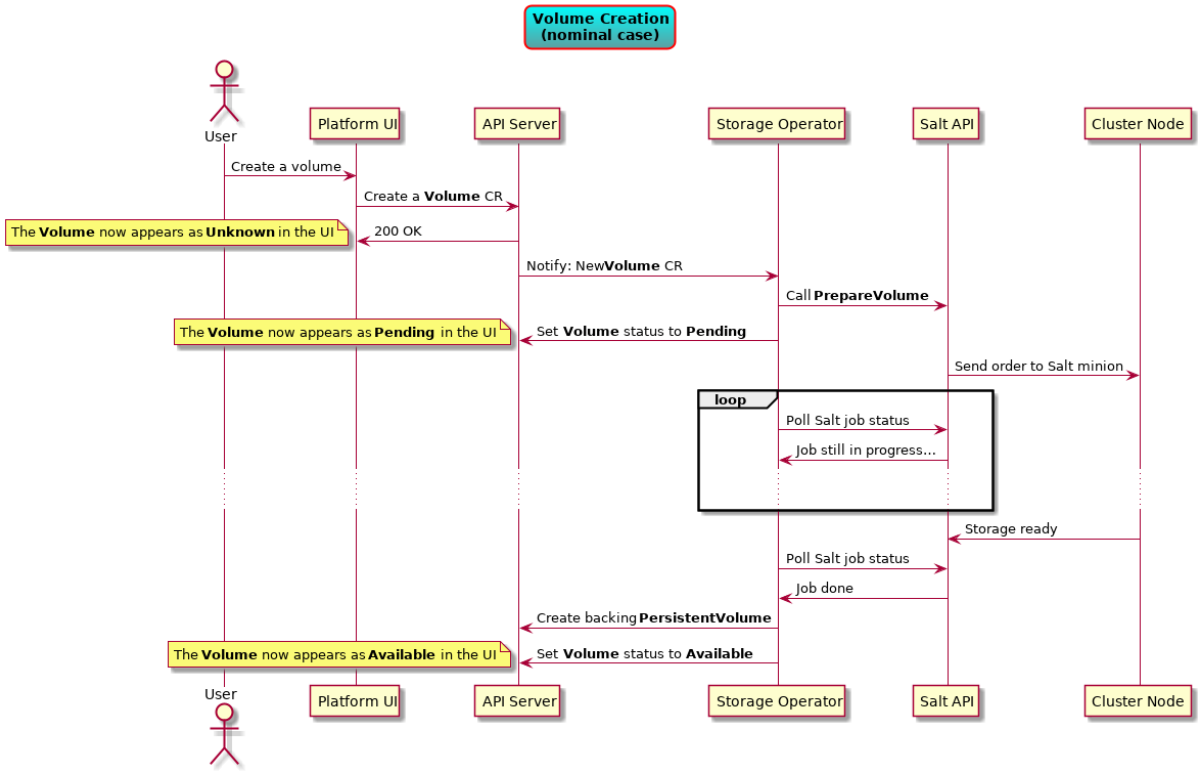
Component Interactions

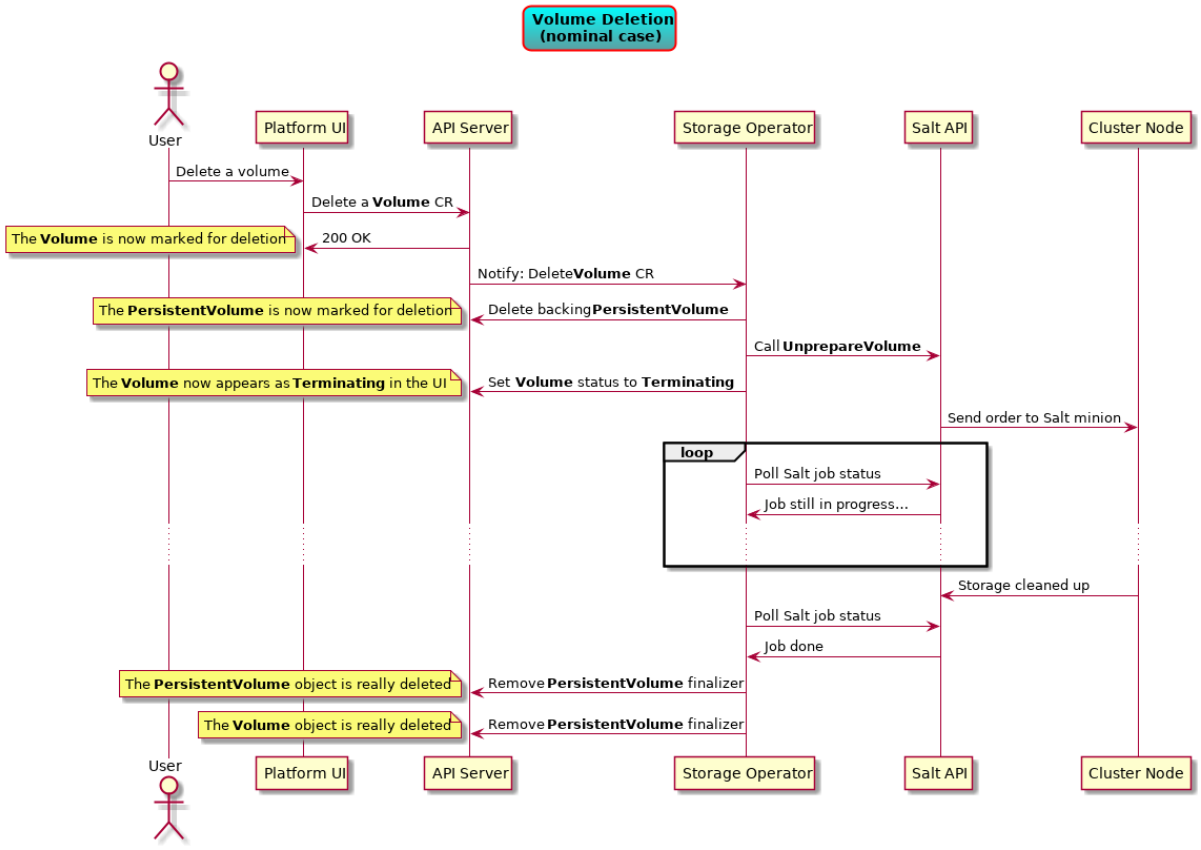
User will create Metalk8s volumes through the Platform UI.

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 though a dedicated Salt account (with limited privileges) using credentials from a dedicated cluster **Service Account**.





14.1.4 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

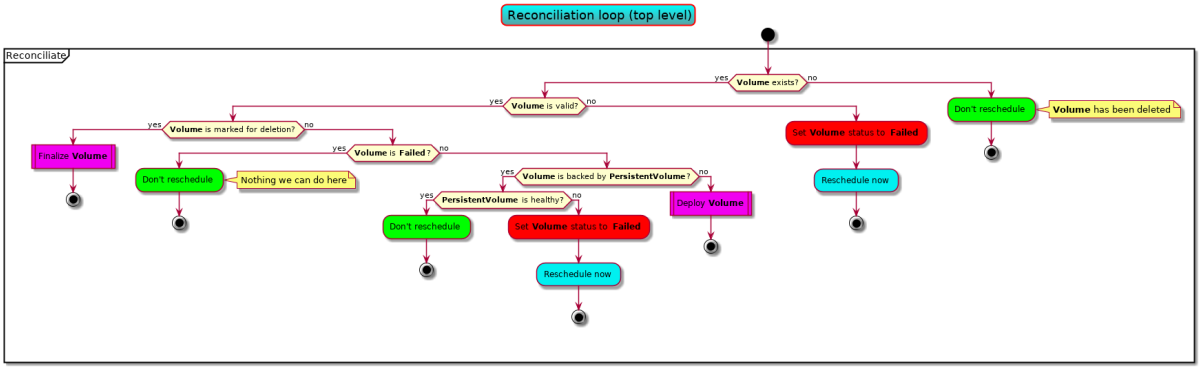
Reconciliation Loop (Top Level)

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 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

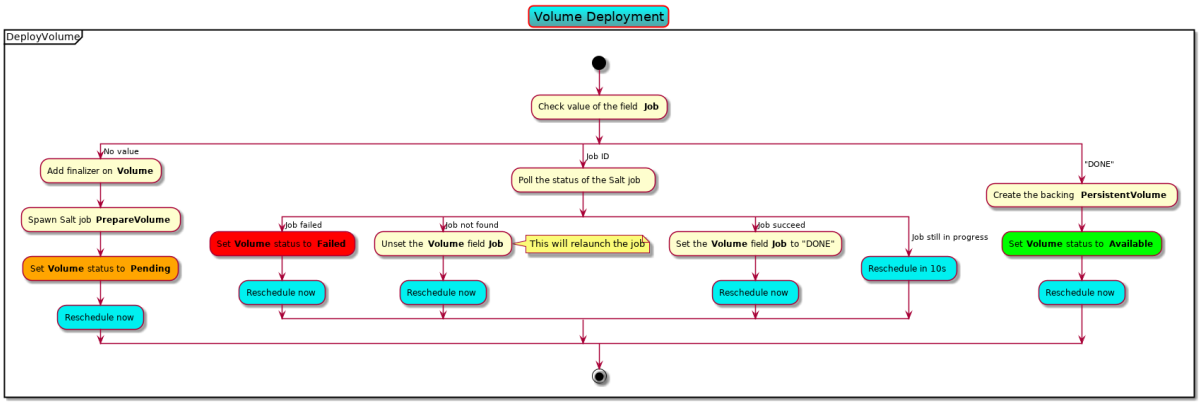
To deploy a volume, the operator needs to prepare its storage (using Salt) and create a backing **PersistentVolume**.

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 successfuly 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).

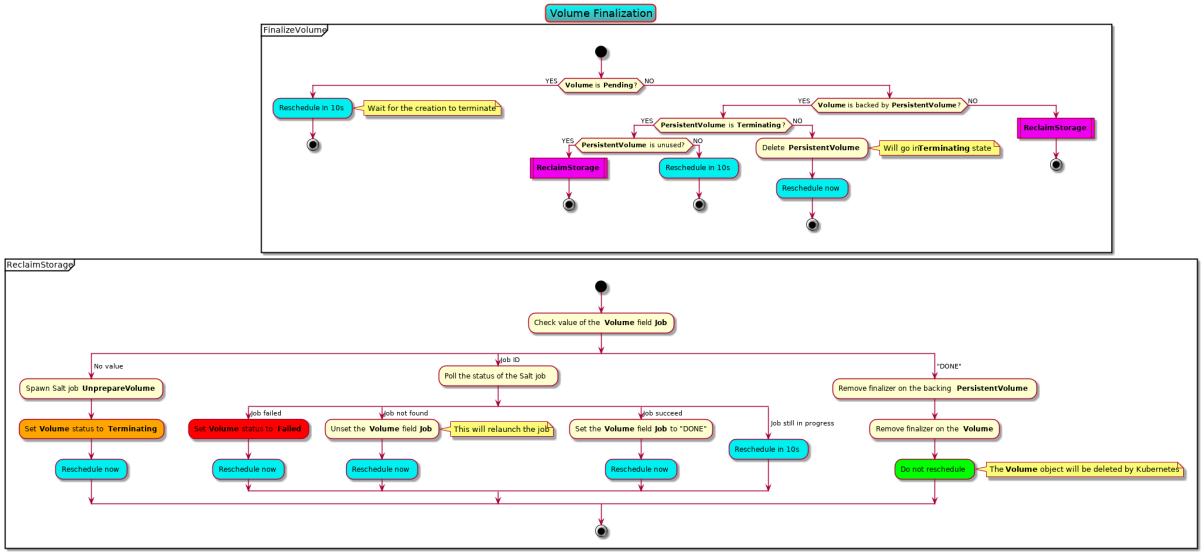


Volume Finalization

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 we 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

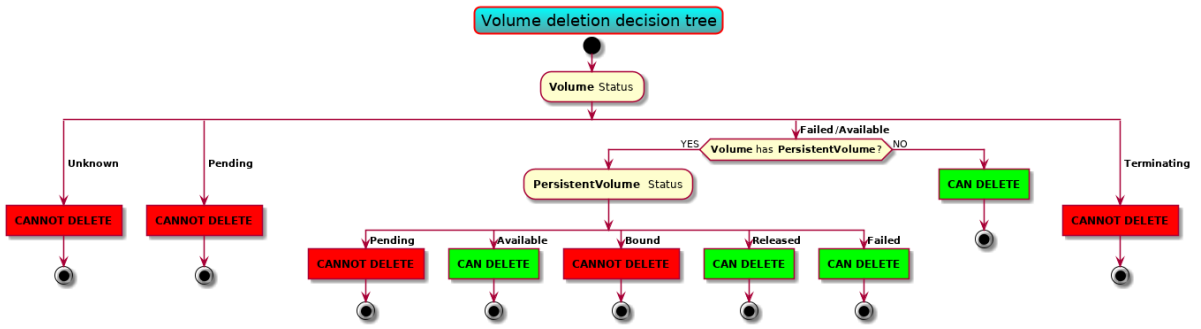
A volume should be deletable from the UI when it's deletable from a user point of view (you can always delete an object from the API), i.e. when deleting the object will trigger an "immediate" deletion (i.e. the object won't be retained).

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**)



14.1.5 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 **VolumeBindingMode** to **WaitForFirstConsumer**)

In the Developper Documentation:

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

14.1.6 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.

CHAPTER

FIFTEEN

HOW TO BUILD METALK8S

15.1 Requirements

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

15.1.1 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
- Go (1.12 or higher) and operator-sdk (0.9 or higher): to build the Kubernetes Operators
- mkisofs: to create the MetalK8s ISO
- Mercurial: some Go dependencies are downloaded from Mercurial repositories.

15.1.2 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

15.1.3 Development

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

15.2 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.

15.2.1 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

15.3 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
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
```

15.4 Buildchain features

Here are some useful doit commands/features, for more information, the official documentation is here.

15.4.1 doit tabcompletion

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

15.4.2 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`.

15.4.3 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`.

15.4.4 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_packages:calico-cni-plugin:pkg_srpm

_build_packages:calico-cni-plugin:pkg_srpm

Build calico-cni-plugin-3.5.1-1.el7.src.rpm

status     : up-to-date

file_dep   :
 - /home/foo/dev/metalk8s/_build/metalk8s-build-latest.tar.gz
 - /home/foo/dev/metalk8s/_build/packages/calico-cni-plugin/SOURCES/v3.5.1.tar.gz
 - /home/foo/dev/metalk8s/_build/packages/calico-cni-plugin/SOURCES/calico-ipam-amd64
 - /home/foo/dev/metalk8s/packages/calico-cni-plugin.spec
 - /home/foo/dev/metalk8s/_build/packages/calico-cni-plugin/SOURCES/calico-amd64

task_dep   :
 - _package_mkdir_root
 - _build_packages:calico-cni-plugin:pkg_mkdir

targets    :
 - /home/foo/dev/metalk8s/_build/packages/calico-cni-plugin-3.5.1-1.el7.src.rpm
```

15.4.5 Wildcard selection

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

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

CHAPTER

SIXTEEN

HOW TO RUN COMPONENTS LOCALLY

16.1 Running a cluster locally

16.1.1 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

16.1.2 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 boostrap configuration
- execute the bootstrap script to make this machine a 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`).

16.2 Running the storage operator locally

16.2.1 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.

16.2.2 Prerequisites

- You should have a running Metalk8s cluster somewhere
- You should have installed the dependencies locally with `cd storage-operator; go mod download`

16.2.3 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.cong-you-copied-locally>
export METALK8S_SALT_MASTER_ADDRESS=https://<ADDRESS-OF-SALT-API>
operator-sdk up local
```

16.3 Running the platform UI locally

16.3.1 Requirements

- Node.js, 10.16

16.3.2 Prerequisites

- You should have a running Metalk8s cluster somewhere
- You should have installed the dependencies locally with `cd ui; npm install`

16.3.3 Procedure

1. Connect to the boostrap 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'
])
pillar = json.loads(output)['local']
ui_conf = {
    'url': 'https://{}:6443'.format(pillar['api_server']['host']),
    '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']
    ),
}
```

(continues on next page)

(continued from previous page)

```
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`

CHAPTER

SEVENTEEN

DEVELOPMENT BEST PRACTICES

17.1 Python best practices

17.1.1 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 unecessary.

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

17.1.2 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 recommandation 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
        # [. . .]
```

17.1.3 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 catched 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!")
```

(continues on next page)

(continued from previous page)

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

References

- hasattr() – A Dangerous Misnomer

CHAPTER

EIGHTEEN

INTEGRATING WITH METALK8S

18.1 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.

18.1.1 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.

18.1.2 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.

18.1.3 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)

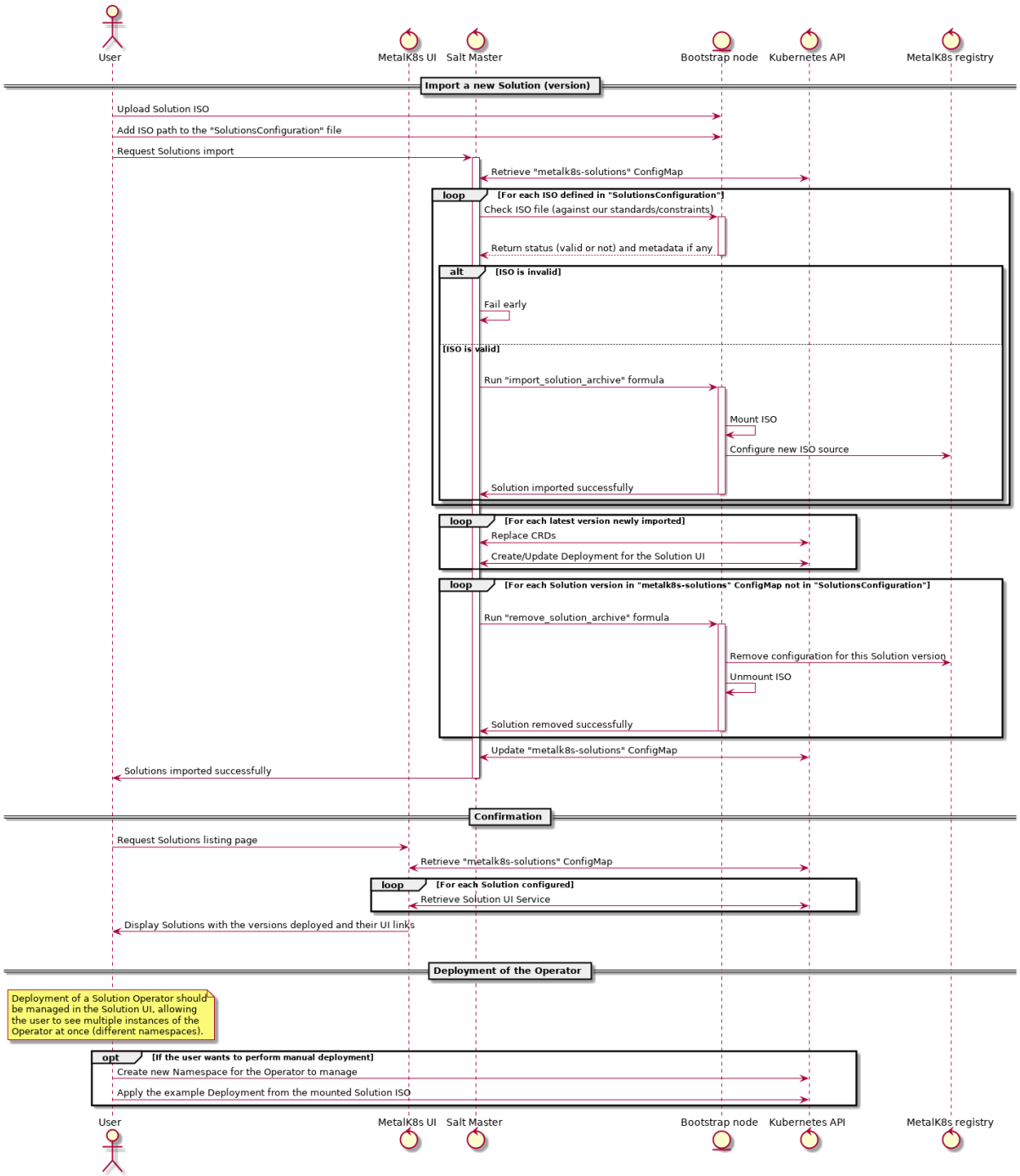
Note: Solutions can leverage the Prometheus Operator CRs for setting up the monitoring of their components. For more information, see *Monitoring* and *Solution Operator guidelines*.

Todo: Define how Solutions can deploy Grafana dashboards.

18.1.4 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.

18.2 Solution archive guidelines

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

18.2.1 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

18.2.2 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
```

18.2.3 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)
```

18.2.4 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*.

18.2.5 Operator

See *Solution Operator guidelines* for how the `/operator` directory should be populated.

18.2.6 Web UI

Todo: Create UI guidelines and reference here

18.3 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*.

18.3.1 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.

18.3.2 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.

18.4 Deploying And Experimenting

Given the solution ISO is correctly generated, a script utiliy has been added to enable solution install and removal

18.4.1 Installation

Use the *solution-manager.sh* script to install a new solution ISO using the following command

```
/src/scality/metalk8s-X.X.X/solution-manager.sh -a/--add </path/to/new/ISO>
```

18.4.2 Removal

To remove a solution from the cluster use the previous script by invoking

```
/src/scality/metalk8s-X.X.X/solution-manager.sh -d/--del </path/to/ISO>
```

Part V

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.

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 *https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/*

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.

`kubectl` `kubectl` is a CLI interface for interacting with a Kubernetes cluster.

See also the official Kubernetes documentation for `kubectl`.

INDEX

A

Alertmanager, 91
API Server, 91

B

Bootstrap, 91
Bootstrap node, 91

C

Controller Manager, 91

E

etcd, 91

K

kube-apiserver, 91
kube-controller-manager, 91
kube-scheduler, 92
Kubeconfig, 91
kubectl, 92
Kubelet, 91

N

Node, 91
Node manifest, 91

P

Pod, 91
Prometheus, 91

S

Salt Master, 91
Salt Minion, 91
SaltAPI, 91
Scheduler, 92
Service, 92

T

Taint, 92
Toleration, 92