# MetalK8s Documentation

*Release 2.0.0-alpha3-1652-g0a2f6a5d-dirty*

**Scality**

**Aug 21, 2020**

# 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 19

II Installation Guide 21

6 Sizing recommendations 23

III Operational Guide 25

7 ISO Preparation 29

8 Upgrade Guide 31

9 Downgrade Guide 33

10 Supported Versions 35

11 Downgrade Pre-requisites 37

12 Downgrade Steps 39

IV Developer Guide 41

13 Architecture Documents 43

14 How to build MetalK8s 55

15 How to run components locally 59

16 Development 61

V Glossary 73

Index 77

# 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:

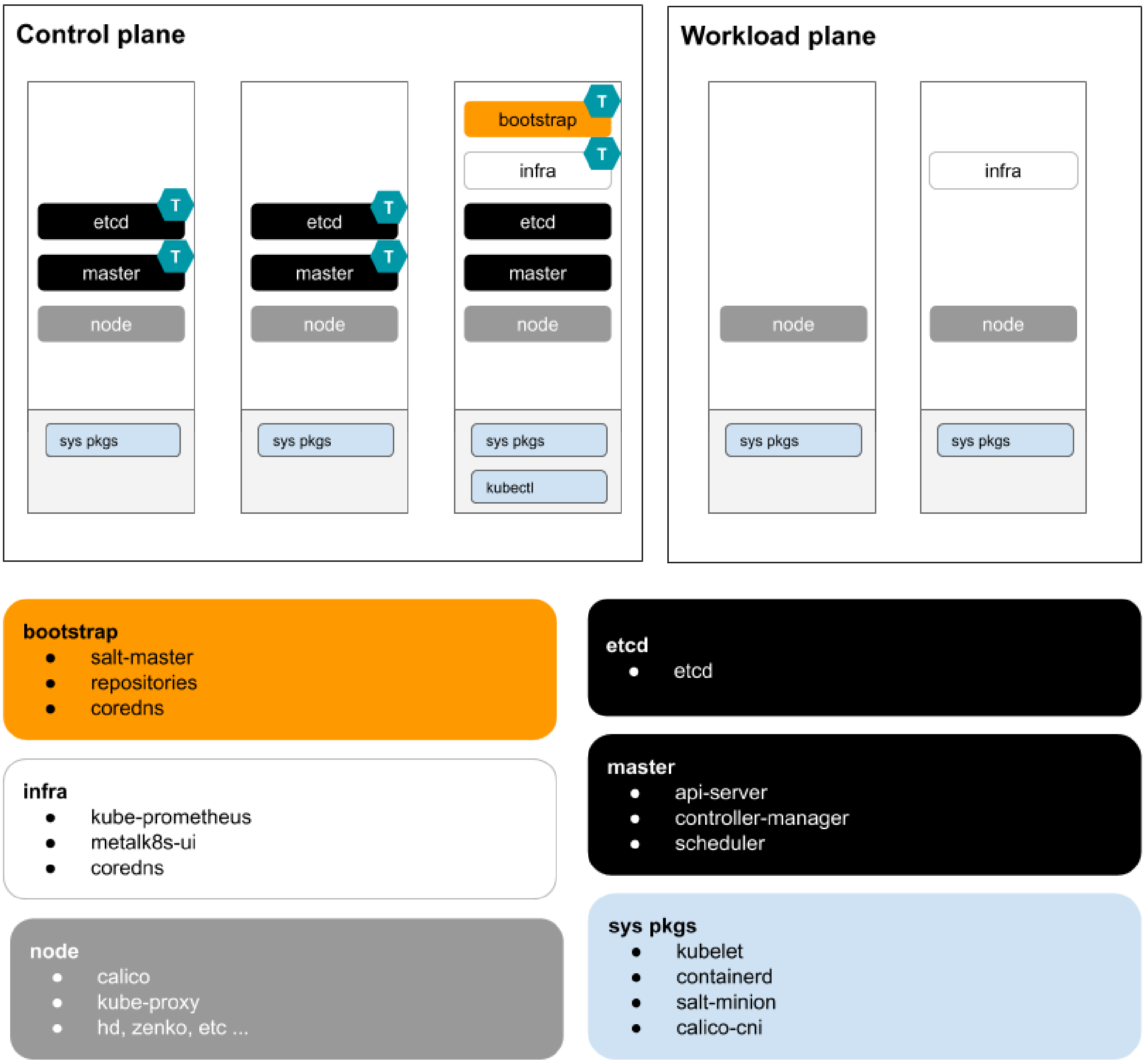
- A network for *Pods*, defaulting to `10.233.0.0/16`
- A network for *Services*, defaulting to `10.96.0.0/12`

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-2.2.0-dev
root@bootstrap $ mount <path-to-iso> /srv/scality/metalk8s-2.2.0-dev
```

## 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-2.2.0-dev/bootstrap.sh
```

### 3.3.2 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>
```

(continues on next page)

(continued from previous page)

```
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>
```

### 3.3.3 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 through *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 from the command-line

### 4.2.1 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: '2.2.0-dev'
    <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:
  [...]
  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.

### 4.2.2 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
```

### 4.2.3 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-2.2.0-dev \
                              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
```

### 4.2.4 Troubleshooting

**Todo:**

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

## 4.3 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 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

## 5.2 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

# 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 EIGHT

# UPGRADE GUIDE

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

## 8.1 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.

## 8.2 Upgrade Pre-requisites

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

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

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

## 8.3 Upgrade Steps

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

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

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

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

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

CHAPTER

NINE

# DOWNGRADE GUIDE

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

CHAPTER

TEN

# SUPPORTED VERSIONS

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

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

Please refer to the release notes for more information.

## CHAPTER ELEVEN

# DOWNGRADE PRE-REQUISITES

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

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

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

CHAPTER

# TWELVE

## DOWNGRADE STEPS

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

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

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

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

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

# Part IV

# Developer Guide

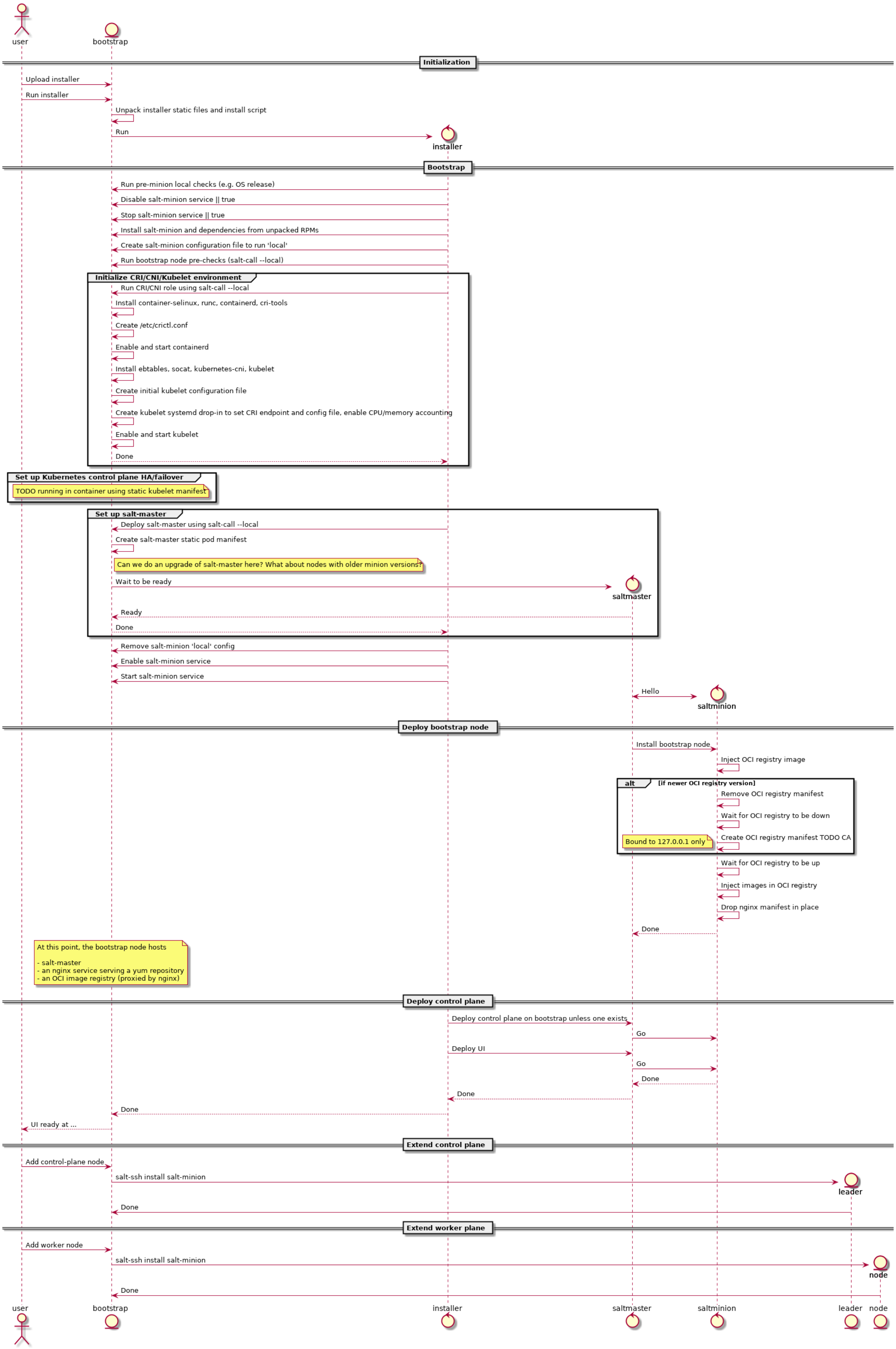
CHAPTER

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 Requirements

### 13.2.1 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.

### 13.2.2 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.

### 13.2.3 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.

### 13.2.4 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

### 13.2.5 Storage

TODO

### 13.2.6 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).

## 13.3 Continuous Testing

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

The goal is to:

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

Lets define 2 differents stages of continuous testing:

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

### 13.3.1 Pre-merge

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

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

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

- Simple upgrade/downgrade

### 13.3.2 Post-merge

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

#### Nightly

- Upgrade, downgrade tests:
  - For previous development branch

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

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

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

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

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

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

**Note:** Complex deployment is (to be validated):

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

**Todo:**

- Describe solutions tests (#1993)

#### Weekly

More complex tests:

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

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

### 13.3.3 Adaptive test plan

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

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

For example:

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

**Todo:** Several questions:

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

CHAPTER

FOURTEEN

# HOW TO BUILD METALK8S

## 14.1 Requirements

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

### 14.1.1 Mandatory

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

### 14.1.2 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

### 14.1.3 Development

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

## 14.2 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.

### 14.2.1 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

## 14.3 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)

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
```

(continues on next page)

(continued from previous page)

```
export SKOPEO_BIN=skopeo
export VAGRANT_BIN=vagrant
```

## 14.4 Buildchain features

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

### 14.4.1 doit tabcompletion

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

### 14.4.2 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`.

### 14.4.3 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`.

### 14.4.4 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

```

(continues on next page)

(continued from previous page)

```
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
```

### 14.4.5 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

FIFTEEN

# HOW TO RUN COMPONENTS LOCALLY

## 15.1 Running a cluster locally

### 15.1.1 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

### 15.1.2 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`).

CHAPTER

SIXTEEN

# DEVELOPMENT

## 16.1 Continuous Testing

### 16.1.1 Add a new test in the continuous integration system

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

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

#### How to choose between Pre-merge and Post-merge

The choice really depends on the goals of this test.

As a high-level view:

Pre-merge:

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

Post-merge:

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

#### How to add a stage in continuous integration system

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

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

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

### A test stage in MetalK8s context

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

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

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

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

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

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

Example:

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

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

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

#### TestRail upload

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

> **Warning:** TestRail upload is only done for Post-merge as we do not want to store each and every test result coming from branches with on-going work.
>
> Do not follow this section if it's not a Post-merge test stage.

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

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

Example:

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

## 16.2 Commit Best Practices

### 16.2.1 How to split a change into commits

#### Why do we need to split changes into commits

This has several advantages amongst which are:

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

#### Examples

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

##### Cosmetic changes

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

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

Example (Pull request #1620):

- one commit for the cosmetic changes: 766f572e462c6933c8168a629ed4f479bb68a803
- one commit for the functional changes: 3367fabdefc0b35d34bf7cf2fb0d33ff81f9fd5a

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

##### Refactoring

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

1. first add a new helper: 29f49cbe9dfa
2. then use it in new code: 7e47310a8f20

Or:

1. first add the new code: 5b2a6d5fa498
2. then refactor the now duplicated code: ac08d0f53a83

##### Mixing unrelated changes

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

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

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

### 16.2.2 How to write a commit message

#### Why do we need commit messages

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

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

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

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

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

### Anatomy of a good commit message

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

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

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

#### Subject

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

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

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

It is composed of:

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

Examples:

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

If several components are affected:

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

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

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

### Body

The body should answer the following questions:

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

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

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

Lines must be wrapped at 72 characters.

### Footer

Use references such as `Refs`, `See`, `Fixes` or `Closes` followed by an issue number to automate issue management.

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

Example:

```
topic: description

[ commit message body ]

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

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

### Examples

#### Bad commit message

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

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

#### Good commit message

##### Commit b531290c04c4

```
Add gzip to nginx conf

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

Some things to note about this commit message:

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

##### Commit 82d92836d4ff

```
Use safer invocation of shell commands

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

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

Issue: GH-781
```

Some things to note about this commit message:

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

#### Commit f66ac0be1c19

```
build: fix concurrent build on MacOS

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

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

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

There are three ways to solve this problems:
1. set the environment variable `OBJC_DISABLE_INITIALIZE_FORK_SAFETY=YES.`
2. don't use `fork`
3. fix the code that uses a dangerous mix of thread and forks

(1) is not good as it doesn't fix the underlying issue: it only disable
the security and we're back to "works most of the time, sometimes does
weird things"
(2) is easy to do because we can tell to `doit` to uses only threads
instead of forks.
(3) is probably the best, but requires more troubleshooting/time/

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

[1]: http://sealiesoftware.com/blog/archive/2017/6/5/Objective-C_and_fork_in_macOS_1013.html
[2]: https://blog.phusion.nl/2017/10/13/why-ruby-app-servers-break-on-macos-high-sierra-and-what-
↪can-be-done-about-it/

Closes: #1354
```

Some things to note about this commit message:

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

### 16.2.3 Conclusion

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

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

## 16.3 Python best practices

### 16.3.1 Import

#### Avoid `from module_foo import symbol_bar`

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

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

All in all, this should be avoided when 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

### 16.3.2 Naming

#### Predicate functions

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

This 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
        # [...]
```

## 16.3.3 Patterns and idioms

### Don’t write code vulnerable to “Time of check to time of use”

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

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

In Python terms:

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

#### Rationale

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

#### Examples

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

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

#### References

- Time of check to time of use

### Minimize the amount of code in a `try` block

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

Indeed, if the `try` block spans over several statements that can raise an exception 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!")

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

### References

- hasattr() – A Dangerous Misnomer

# 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.

**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, 75
API Server, 75

B

Bootstrap, 75
Bootstrap node, 75

C

Controller Manager, 75

E

etcd, 75

K

kube-apiserver, 75
kube-controller-manager, 75
kube-scheduler, 76
Kubeconfig, 75
kubectl, 76

N

Node, 75
Node manifest, 75

P

Pod, 75
Prometheus, 75

S

Salt Master, 75
Salt Minion, 75
SaltAPI, 75
Scheduler, 75
Service, 76

T

Taint, 76
Toleration, 76