MetalK8s

Scality

May 23, 2022

CONTENTS:

1 Installation **3**
- 1.1 Introduction . . . 3
- 1.2 Prerequisites . . . 9
- 1.3 Deployment of the Bootstrap node . . . 12
- 1.4 Enable IP-in-IP Encapsulation . . . 17
- 1.5 Cluster expansion . . . 17
- 1.6 Post-Installation Procedure . . . 24
- 1.7 Accessing Cluster Services . . . 29
- 1.8 Advanced guide . . . 31
- 1.9 Troubleshooting . . . 31

2 Operation **35**
- 2.1 Cluster Monitoring . . . 35
- 2.2 Account Administration . . . 39
- 2.3 Cluster and Services Configurations . . . 44
- 2.4 Volume Management . . . 63
- 2.5 Cluster Upgrade . . . 69
- 2.6 Cluster Downgrade . . . 70
- 2.7 Disaster Recovery . . . 71
- 2.8 Solution Deployment . . . 73
- 2.9 Changing the hostname of a MetalK8s node . . . 74
- 2.10 Changing the Control Plane Ingress IP . . . 76
- 2.11 Using the `metalk8s-utils` Image . . . 77
- 2.12 Registry HA . . . 78
- 2.13 Listening Processes . . . 79
- 2.14 Troubleshooting . . . 80
- 2.15 Sosreport . . . 83

3 Developer Guide **85**
- 3.1 Architecture Documents . . . 85
- 3.2 How to build MetalK8s . . . 179
- 3.3 How to run components locally . . . 183
- 3.4 Deploy new MetalK8s image . . . 186
- 3.5 Development . . . 187
- 3.6 Integrating with MetalK8s . . . 222
- 3.7 Shared Tooling . . . 233

4 Glossary **241**

5 Indices and tables **245**

Python Module Index 247

Index 249

MetalK8s is an opinionated Kubernetes distribution with a focus on long-term on-prem deployments, launched by Scality to deploy its Zenko solution in customer datacenters.

See the *Installation* to begin installing a MetalK8s cluster.

CHAPTER

ONE

INSTALLATION

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

1.1 Introduction

1.1.1 Foreword

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

Installing a MetalK8s cluster can be broken down into the following steps:

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

Warning: MetalK8s is not designed to handle world-distributed multi-site architectures. Instead, it provides a highly resilient cluster at the datacenter scale. To manage multiple sites, look into application-level solutions or alternatives from such Kubernetes community groups as the Multicluster SIG).

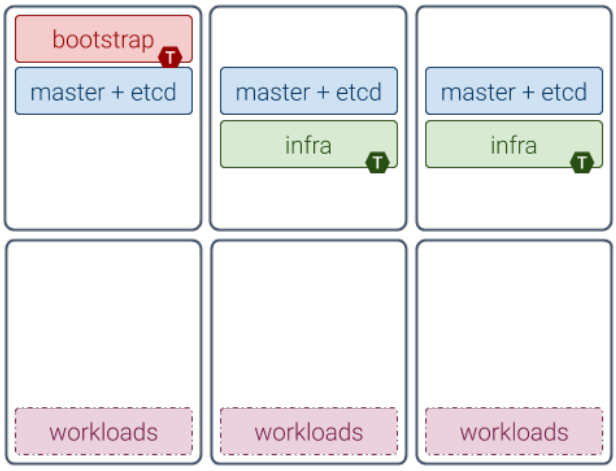
1.1.2 Choosing a Deployment Architecture

Before starting the installation, it's best to choose an architecture.

Standard Architecture

The recommended architecture when installing a small MetalK8s cluster emphasizes ease of installation, while providing high stability for scheduled workloads. This architecture includes:

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



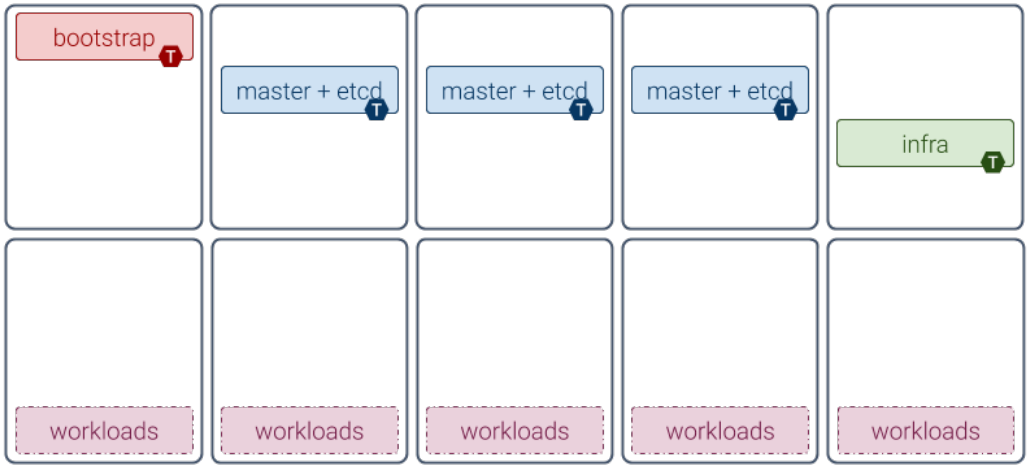
Machines dedicated to the control plane do not require many resources (see *the sizing notes below*), and can safely run as virtual machines. Running workloads on dedicated machines makes them simpler to size, as MetalK8s impact will be negligible.

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

Extended Architecture

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

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

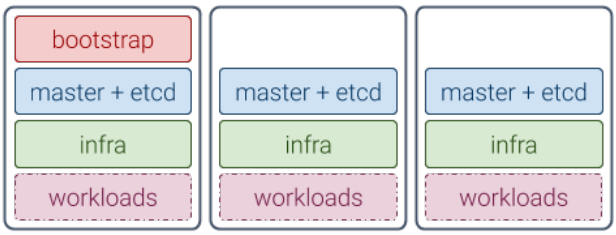


Compact Architectures

Although its design is not focused on having the smallest compute and memory footprints, MetalK8s can provide a fully functional single-node "cluster". The bootstrap node can be configured to also allow running applications next to all other required services (see *the section about taints* below).

Because a single-node cluster has no resilience to machine or site failure, a three-machine cluster is the most compact recommended production architecture. This architecture includes:

- Two machines running control plane services alongside infra and workload applications
- One machine running bootstrap services and all other services



Note: Sizing for such compact clusters must account for the expected load. The exact impact of colocating an application with MetalK8s services must be evaluated by that application's provider.

Variations

You can customize your architecture using combinations of *roles* and *taints*, described below, to adapt to the available infrastructure.

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

You can also evolve an architecture after initial deployment, if the underlying infrastructure also evolves (new machines can be added through the *expansion* mechanism, roles can be added or removed, etc.).

1.1.3 Concepts

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

Nodes

Nodes are Kubernetes worker machines that allow running containers and can be managed by the cluster (see *control plane services*, next section).

Control and Workload Planes

The distinction between the control and workload planes 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 of a Kubernetes control plane are:

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

The **workload plane** is the set of nodes in which applications are deployed via Kubernetes objects, managed by services in 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).

Node Roles

A node's responsibilities are determined using roles. Roles are stored in *Node manifests* using labels of the form `node-role.kubernetes.io/<role-name>: ''`.

MetalK8s uses five different roles, which may be combined freely:

`node-role.kubernetes.io/master` The master role marks a control plane member. *Control plane services* can only be scheduled on master nodes.

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

`node-role.kubernetes.io/infra` The infra role is specific to MetalK8s. It marks nodes where non-critical cluster services (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 *Pod* images
- A *Salt Master* and its associated *SaltAPI*

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

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

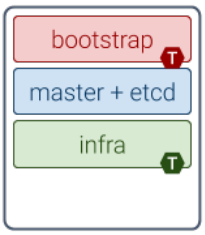
Node Taints

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

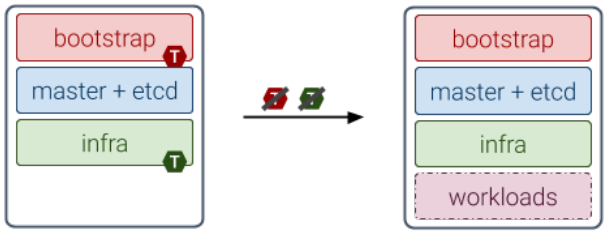
Taints allow dedicating Nodes to specific use cases, such as running control plane services.

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

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



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



To do this, use the following commands after deployment:

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

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

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 enables cluster services to communicate with each other. The workload plane network exposes applications, including those in infra Nodes, to the outside world.

MetalK8s also enables configuring virtual networks for internal communication:

- 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 existing infrastructure, choose other ranges during *Bootstrap configuration*.

1.1.4 Additional Notes

Sizing

Sizing the machines in a MetalK8s cluster depends on the selected architecture and anticipated changes. Refer to the documentation of the applications planned to run in the deployed cluster before completing the sizing, as their needs will compete with the cluster's.

Each *role*, describing a group of services, requires a certain amount of resources to run properly. If multiple roles are used on a single Node, these requirements add up.

Role	Services	CPU	RAM	Required Storage	Recommended Storage
bootstrap	Package repositories, container registries, Salt master	1 core	2 GB	Sufficient space for the product ISO archives	
etcd	etcd database for the K8s API	0.5 core	1 GB	1 GB for /var/lib/etcd	
master	K8s API, scheduler, and controllers	0.5 core	1 GB		
infra	Monitoring services, Ingress controllers	0.5 core	2 GB	10 GB partition for Prometheus 1 GB partition for Alertmanager	
requirements common to any Node	Salt minion, Kubelet	0.2 core	0.5 GB	**40 GB root partition**	100 GB or more for /var

These numbers do not account for highly unstable workloads or other sources of unpredictable load on the cluster services. Providing a safety margin of an additional 50% of resources is recommended.

Consider the official recommendations for etcd sizing, as the stability of a MetalK8s installation depends on the stability of the backing etcd (see the *etcd* section for more details). Prometheus and Alertmanager also require storage, as explained in *Provision Storage for Services*.

Deploying with Cloud Providers

When installing in a virtual environment, such as AWS EC2 or OpenStack, adjust network configurations carefully: virtual environments often add a layer of security at the port level, which must be disabled or circumvented with *IP-in-IP encapsulation*.

Also note that Kubernetes has numerous integrations with existing cloud providers to provide easier access to proprietary features, such as load balancers. For more information, review this topic.

1.2 Prerequisites

MetalK8s clusters require machines running CentOS/RHEL 7.6 or higher as their operating system. These machines may be virtual or physical, with no difference in setup procedure. The number of machines to set up depends on the architecture you chose in *Choosing a Deployment Architecture*.

Machines *must not* be managed by any configuration management system, such as SaltStack or Puppet.

> **Warning:** The distribution must be left intact as much as possible (do not tune, tweak, or configure it, or install any software).

1.2.1 Proxies

For nodes operating behind a proxy, see *Configuration*.

1.2.2 Linux Kernel Version

Linux kernels shipped with CentOS/RHEL 7 and earlier are affected by a cgroups memory leak bug.

This bug was fixed in kernel 3.10.0-1062.4.1. Use this kernel version or later.

The version can be retrieved using:

```
$ uname -r
```

If the installed version is lower than the one above, upgrade it with:

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

These commands may require sudo or root access.

1.2.3 Provisioning

SSH

Each machine must be accessible through SSH from the host. *Bootstrap node deployment* generates a new SSH identity for the *Bootstrap node* and shares it with other nodes in the cluster. You can also do this manually beforehand.

Network

Each machine must be a member of both the control plane and workload plane networks described in *Networks*. However, these networks can overlap, and nodes do not need distinct IP addresses for each plane.

For the host to reach the cluster-provided UIs, it must be able to connect to the machines' control plane IP addresses.

Repositories

Each machine must have properly configured repositories with access to basic repository packages (depending on the operating system).

CentOS:

- base
- extras
- updates

RHEL 7:

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

RHEL 8:

- rhel-8-for-x86_64-baseos-rpms
- rhel-8-for-x86_64-appstream-rpms

Note: RHEL instances must be registered.

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

To enable an existing repository:

CentOS:

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

RHEL:

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

To add a new repository:

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

Note: *repo_url* can be set to a remote URL using the prefix *http://*, *https://*, *ftp://*, etc., or to a local path using *file://*.

For more, review the official Red Hat documentation:

- Enable Optional repositories with RHSM
- Configure repositories with YUM
- Advanced repositories configuration

etcd

For production environments, a block device dedicated to *etcd* is recommended for better performance and stability. With lower write latency and less variance than spinning disks, SSDs are recommended to improve reliability.

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

For more on etcd's hardware requirements, see the official documentation.

1.3 Deployment of the Bootstrap node

1.3.1 Preparation

1. Retrieve a MetalK8s ISO (you may build one yourself by following our developer guide). Scality customers can retrieve validated builds as part of their license from the Scality repositories.
2. Download the MetalK8s ISO file on the machine that will host the bootstrap node. Run *checkisomd5 –verbose <path-to-iso>* to validate its integrity (*checkisomd5* is part of the *isomd5sum* package).
3. Mount this ISO file at the path of your choice (we will use `/srv/scality/metalk8s-|version|` for the rest of this guide, as this is where the ISO will be mounted automatically after running the bootstrap script):

   ```
   root@bootstrap $ mkdir -p /srv/scality/metalk8s-2.11.7
   root@bootstrap $ mount <path-to-iso> /srv/scality/metalk8s-2.11.7
   ```

1.3.2 Configuration

1. Create the MetalK8s configuration directory.

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

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

   ```
   apiVersion: metalk8s.scality.com/v1alpha3
   kind: BootstrapConfiguration
   networks:
     controlPlane:
       cidr: <CIDR-notation>
       ingress:
         ip: <IP-for-ingress>
         controller:
           replicas: 2
           affinity:
             podAntiAffinity:
               hard: []
               soft:
                 - topologyKey: kubernetes.io/hostname
       metalLB:
         enabled: <boolean>
     workloadPlane:
       cidr: <CIDR-notation>
       mtu: <network-MTU>
     pods: <CIDR-notation>
     services: <CIDR-notation>
   proxies:
     http: <http://proxy-ip:proxy-port>
     https: <https://proxy-ip:proxy-port>
     no_proxy:
       - <host>
       - <ip/cidr>
   ```

(continues on next page)

(continued from previous page)

```
ca:
  minion: <hostname-of-the-bootstrap-node>
archives:
  - <path-to-metalk8s-iso>
kubernetes:
  apiServer:
    featureGates:
      <feature_gate_name>: True
  controllerManager:
    config:
      terminatedPodGCThreshold: 500
  coreDNS:
    replicas: 2
    affinity:
      podAntiAffinity:
        hard: []
        soft:
          - topologyKey: kubernetes.io/hostname
```

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

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

Note: Several CIDRs can be provided if all nodes do not sit in the same network. This is an *advanced configuration* which we do not recommend for non-experts.

For `controlPlane` entry, an `ingress` can also be provided. This section allow to set the IP that will be used to connect to all the control plane components, like MetalK8s-UI and the whole monitoring stack. We suggest using a Virtual IP that will sit on a working master Node. The default value for this Ingress IP is the control plane IP of the Bootstrap node (which means that if you lose the Bootstrap node, you no longer have access to any control plane component).

If you want to override the default `controlPlane ingress` controller podAntiAffinity or number of replicas, by default MetalK8s deploy 2 replicas and use soft podAntiAffinity on hostname so that if it's possible those controllers pods will be spread on different `master` nodes.

Note: Affinity and number of replicas for control plane ingress controller will be ignored if `MetalLB` is disabled, as this control plane ingress controller will be deployed as a DaemonSet, which means that a pod will run on every `master` nodes by default.

This `ip` for `ingress` can be managed by MetalK8s directly if it's possible in your environment, to do so we use MetalLB that allow to manage this Virtual IP directly on Layer2 using only ARP requests, in order to be able to use MetalLB your network need to properly broadcast ARP requests so that Control Plane node hosting the Virtual IP can answer to this ARP request. When MetalLB is enabled this ingress IP is mandatory.

For `workloadPlane` entry an MTU can also be provided, this MTU value should be the lowest MTU value accross all the workload plane network. The default value for this MTU is 1460.

```
networks:
  controlPlane:
```

(continues on next page)

(continued from previous page)

```
    cidr: 10.200.1.0/28
  workloadPlane:
    cidr: 10.200.1.0/28
    mtu: 1500
```

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

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

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

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

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

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

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

The `kubernetes` field can be omitted if you do not have any specific Kubernetes Feature Gates to enable or disable and if you are ok with defaults kubernetes configuration.

If you need to enable or disable specific features for `kube-apiserver` configure the corresponding entries in the `kubernetes.apiServer.featureGates` mapping.

If you want to override the default `coreDNS` podAntiAffinity or number of replicas, by default MetalK8s deploy 2 replicas and use soft podAntiAffinity on hostname so that if it's possible `coreDNS` pods will be spread on different infra nodes. If you have more infra node than `coreDNS` replicas, you should set hard podAntiAffinity on hostname so that you are sure that `coreDNS` pods sit on different node, to do so:

```
kubernetes:
  coreDNS:
    affinity:
      podAntiAffinity:
        hard:
          - topologyKey: kubernetes.io/hostname
```

From `controllerManager` section you can override the number of terminated pods that can exist before the terminated pod garbage collector starts deleting them. If it's set to 0, the terminated pod garbage collector is disabled (default to `500`)

1.3.3 SSH Provisioning

1. Prepare the MetalK8s PKI directory.

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

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

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

> **Warning:** Although the key name is not critical (will be re-used afterwards, so make sure to replace 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).
   1. Retrieve the public key from the Bootstrap node.

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

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

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

1.3.4 Installation

Run the Installation

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

```
root@bootstrap $ /srv/scality/metalk8s-2.11.7/bootstrap.sh
```

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

Validate the install

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

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

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

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

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

```
root@bootstrap $ kubectl get nodes --kubeconfig /etc/kubernetes/admin.conf
NAME                    STATUS    ROLES                          AGE       VERSION
bootstrap               Ready     bootstrap,etcd,infra,master    17m       v1.15.5

root@bootstrap $ kubectl get pods --all-namespaces -o wide --kubeconfig /etc/kubernetes/
↪admin.conf
NAMESPACE             NAME                                                        READY␣
↪STATUS    RESTARTS   AGE     IP               NODE            NOMINATED NODE␣
↪READINESS GATES
kube-system           calico-kube-controllers-7c9944c5f4-h9bsc                      1/1␣
↪Running   0          6m29s   10.233.220.129   bootstrap   <none>           <none>
kube-system           calico-node-v4qhb                                             1/1␣
↪Running   0          6m29s   10.200.3.152     bootstrap   <none>           <none>
kube-system           coredns-ff46db798-k54z9                                       1/1␣
↪Running   0          6m29s   10.233.220.134   bootstrap   <none>           <none>
kube-system           coredns-ff46db798-nvmjl                                       1/1␣
↪Running   0          6m29s   10.233.220.132   bootstrap   <none>           <none>
kube-system           etcd-bootstrap                                                1/1␣
↪Running   0          5m45s   10.200.3.152     bootstrap   <none>           <none>
kube-system           kube-apiserver-bootstrap                                      1/1␣
↪Running   0          5m57s   10.200.3.152     bootstrap   <none>           <none>
kube-system           kube-controller-manager-bootstrap                             1/1␣
↪Running   0          7m4s    10.200.3.152     bootstrap   <none>           <none>
kube-system           kube-proxy-n6zgk                                              1/1␣
↪Running   0          6m32s   10.200.3.152     bootstrap   <none>           <none>
kube-system           kube-scheduler-bootstrap                                      1/1␣
↪Running   0          7m4s    10.200.3.152     bootstrap   <none>           <none>
kube-system           repositories-bootstrap                                        1/1␣
↪Running   0          6m20s   10.200.3.152     bootstrap   <none>           <none>
kube-system           salt-master-bootstrap                                         2/2␣
↪Running   0          6m10s   10.200.3.152     bootstrap   <none>           <none>
kube-system           storage-operator-7567748b6d-hp7gq                             1/1␣
↪Running   0          6m6s    10.233.220.138   bootstrap   <none>           <none>
metalk8s-ingress      nginx-ingress-control-plane-controller-5nkkx                  1/1␣
↪Running   0          6m6s    10.233.220.137   bootstrap   <none>           <none>
metalk8s-ingress      nginx-ingress-controller-shg7x                                1/1␣
↪Running   0          6m7s    10.233.220.135   bootstrap   <none>           <none>
metalk8s-ingress      nginx-ingress-default-backend-7d8898655c-jj7l6                1/1␣
↪Running   0          6m7s    10.233.220.136   bootstrap   <none>           <none>
metalk8s-logging      loki-0                                                        0/1␣
↪Pending   0          6m21s    <none>           <none>      <none>           <none>
metalk8s-monitoring   alertmanager-prometheus-operator-alertmanager-0               0/2␣
↪Pending   0          6m1s    <none>           <none>      <none>           <none>
metalk8s-monitoring   prometheus-operator-grafana-775fbb5b-sgngh                    2/2␣
↪Running   0          6m17s   10.233.220.130   bootstrap   <none>           <none>
metalk8s-monitoring   prometheus-operator-kube-state-metrics-7587b4897c-tt79q       1/1␣
↪Running   0          6m17s   10.233.220.131   bootstrap   <none>           <none>
metalk8s-monitoring   prometheus-operator-operator-7446d89644-zqdlj                 1/1␣
↪Running   0          6m17s   10.233.220.133   bootstrap   <none>           <none>
metalk8s-monitoring   prometheus-operator-prometheus-node-exporter-rb969            1/1␣
↪Running   0          6m17s   10.200.3.152     bootstrap   <none>           <none>
```

(continues on next page)

(continued from previous page)

```
metalk8s-monitoring   prometheus-prometheus-operator-prometheus-0                0/3     ␣
↪Pending   0          5m50s   <none>           <none>       <none>           <none>
metalk8s-ui           metalk8s-ui-6f74ff4bc-fgk86                                 1/1     ␣
↪Running   0          6m4s    10.233.220.139   bootstrap    <none>           <none>
```

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

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

- If you encounter an error during installation or have issues validating a fresh MetalK8s installation, refer to the *Troubleshooting section*.

1.4 Enable IP-in-IP Encapsulation

By default, Calico in MetalK8s is configured to use IP-in-IP encapsulation only for cross-subnet communication.

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

To configure IP-in-IP encapsulation for all communications, run the following command:

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

For more information refer to IP-in-IP Calico configuration.

1.5 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*.

1.5.1 Defining an Architecture

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

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

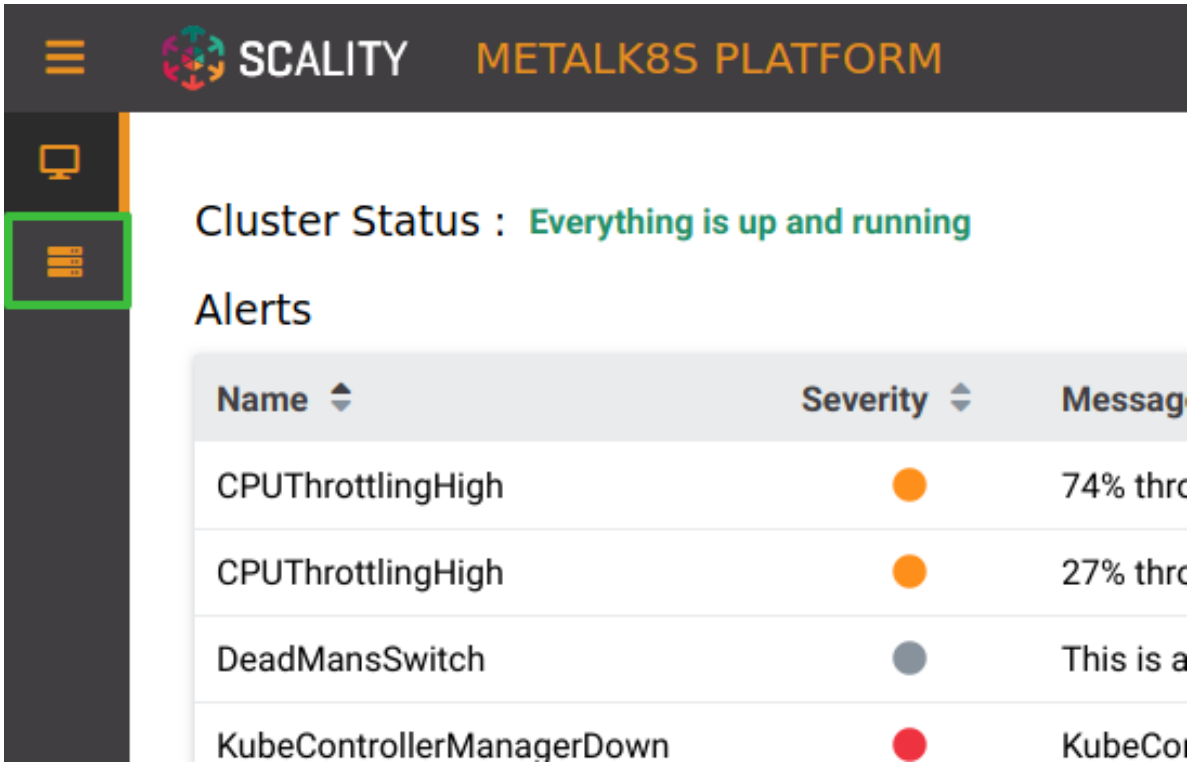
1.5.2 Adding a Node with the MetalK8s GUI

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

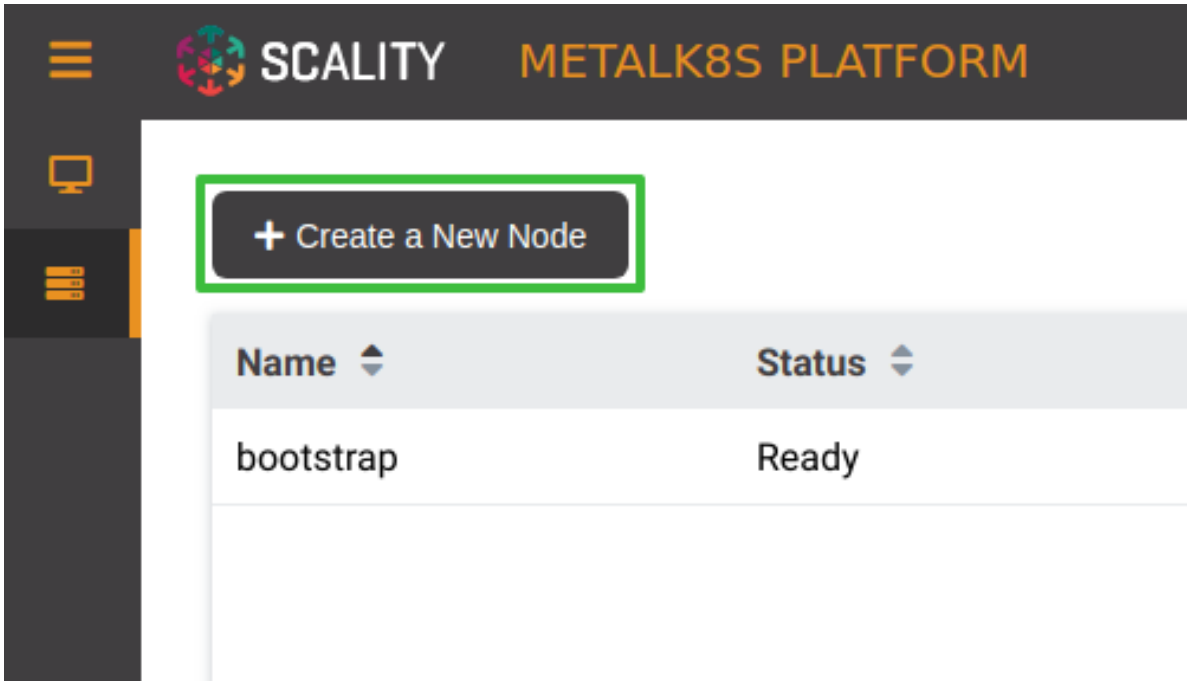
Creating a Node Object

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:



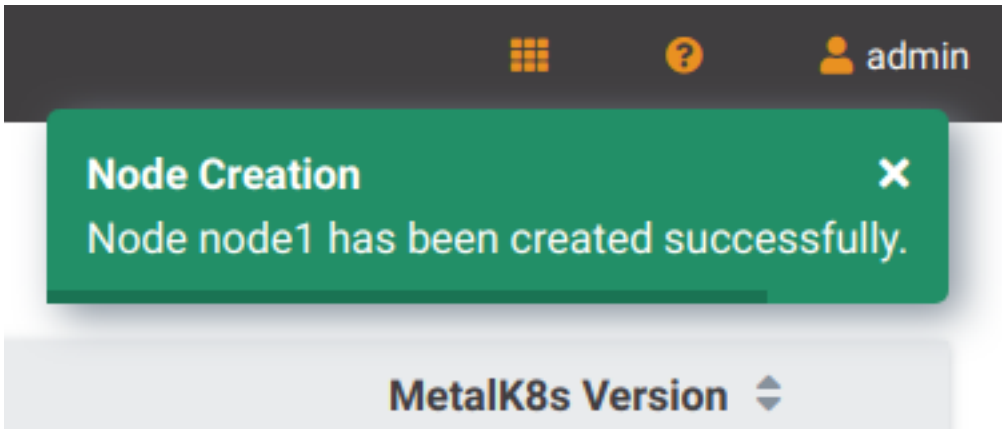
2. From the Node list (the Bootstrap node should be visible there), click the button labeled “Create a New Node”:



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
   - **SSH User**: the user for which the Bootstrap has SSH access
   - **Hostname or IP**: the address to use for SSH from the Bootstrap
   - **SSH Port**: the port to use for SSH from the Bootstrap
   - **SSH Key Path**: the path to the private key generated in *this procedure*
   - **Sudo required**: whether the SSH deployment will need `sudo` access
   - **Roles/Workload Plane**: enable any workload applications run on this Node
   - **Roles/Control Plane**: enable master and etcd services run on this Node
   - **Roles/Infra**: enable infra services run on this Node

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

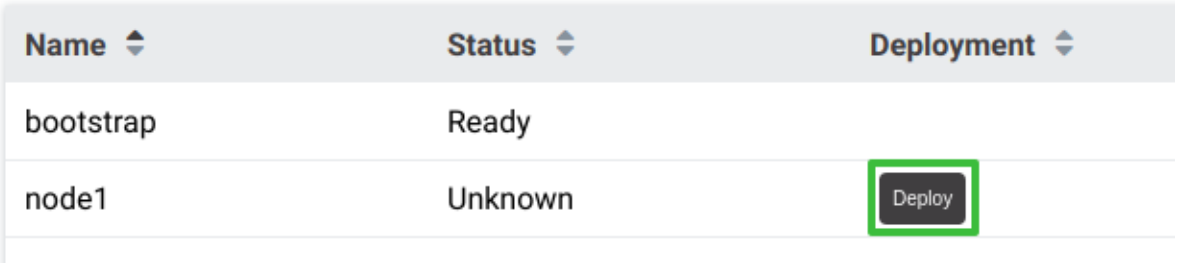
4. Click **Create**. You will be redirected to the Node list page, and will be shown a notification to confirm the Node creation:



Deploying the Node

After the desired state has been declared, it can be applied to the machine. The MetalK8s GUI uses *SaltAPI* to orchestrate the deployment.

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



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



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

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

1.5.3 Adding a Node from the Command-line

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'
    metalk8s.scality.com/ssh-user: 'root'
  labels:
    metalk8s.scality.com/version: '2.11.7'
    <role labels>
spec:
  taints: <taints>
```

Annotations are used by Salt-SSH to connect to the node and deploy it. All annotations are prefixed with `metalk8s.scality.com/`:

Annotation	Description	Default
ssh-host	Control plane IP of the node, must be accessible over SSH from the Bootstrap node	None
ssh-key-path	Path to the private SSH key used to connect to the node	None
ssh-sudo	Whether to use `sudo` to execute commands or not (root privileges are needed to deploy a node, it must be set to `true` if ssh-user is not `root`)	`false`
ssh-user	User to connect to the node and run commands	`root`

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

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

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

  roles and taints both are set to master and etcd. It has the same behavior as the **Control Plane** checkbox in the GUI.

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

(continues on next page)

(continued from previous page)

```
  - effect: NoSchedule
    key: node-role.kubernetes.io/etcd
```

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

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

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

- A simple worker still accepting `infra` services would use the same role label without the taint

  roles are set to node and infra. It's the same as the checkbox of Workload Plane and Infra in MetalK8s GUI.

CLI-only actions

- A Node dedicated to etcd

  roles and taints both are set to etcd.

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

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
```

Deploying the Node

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

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

   Note: Salt SSH requires Python 3 to be installed on the remote host to run Salt functions. It will be installed automatically when deploying the node, though you can send raw shell commands before (using `--raw-shell`) if needed.

   ```
   root@salt-master-bootstrap $ salt-ssh --roster=kubernetes <node_name> --raw-shell
   ↪'echo OK'
   <node_name>:
       ----------
       retcode:
           0
       stderr:
           Warning: Permanently added '<ip>' (ECDSA) to the list of known hosts.
       stdout:
           OK
   ```

2. Start the node deployment.

   ```
   root@salt-master-bootstrap $ salt-run state.orchestrate metalk8s.orchestrate.deploy_
   ↪node \
                               saltenv=metalk8s-2.11.7 \
                               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
   ```

1.5.4 Checking Cluster Health

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

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

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

  https://<first-node-ip>:2379 is healthy: successfully committed proposal: took = 16.
↪285672ms
  https://<second-node-ip>:2379 is healthy: successfully committed proposal: took = 43.
↪462092ms
  https://<third-node-ip>:2379 is healthy: successfully committed proposal: took = 52.
↪879358ms
```

1.6 Post-Installation Procedure

1.6.1 Provision Storage for Services

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

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

To create the required *Volume* objects, use one of the following volume type depending on the platform.

rawBlockDevice Volumes

Write a YAML file with the following contents, replacing:

- <node_name> with the name of the *Node* on which
- <device_path> with the /dev/ path for the partitions to use

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

(continues on next page)

(continued from previous page)

```
    metadata:
      labels:
        app.kubernetes.io/name: 'prometheus-operator-prometheus'
---
apiVersion: storage.metalk8s.scality.com/v1alpha1
kind: Volume
metadata:
  name: <node_name>-alertmanager
spec:
  nodeName: <node_name>
  storageClassName: metalk8s
  rawBlockDevice:  # Choose a device with at least 1GiB capacity
    devicePath: <device_path2>
  template:
    metadata:
      labels:
        app.kubernetes.io/name: 'prometheus-operator-alertmanager'
---
apiVersion: storage.metalk8s.scality.com/v1alpha1
kind: Volume
metadata:
  name: <node_name>-loki
spec:
  nodeName: <node_name>
  storageClassName: metalk8s
  rawBlockDevice:  # Choose a device with at least 10GiB capacity
    devicePath: <device_path3>
  template:
    metadata:
      labels:
        app.kubernetes.io/name: 'loki'
```

lvmLogicalVolume Volumes

Write a YAML file with the following contents, replacing:

- <node_name> with the name of the *Node* on which
- <vg_name> with the existing LVM VolumeGroup name on this specific Node

```
---
apiVersion: storage.metalk8s.scality.com/v1alpha1
kind: Volume
metadata:
  name: <node_name>-prometheus
spec:
  nodeName: <node_name>
  storageClassName: metalk8s
  lvmLogicalVolume:
    vgName: <vg_name>  # Choose an existing LVM VolumeGroup
    size: 10Gi  # Prometheus LogicalVolume should have at least 10GiB capacity
  template:
```

(continues on next page)

(continued from previous page)

```
    metadata:
      labels:
        app.kubernetes.io/name: 'prometheus-operator-prometheus'
---
apiVersion: storage.metalk8s.scality.com/v1alpha1
kind: Volume
metadata:
  name: <node_name>-alertmanager
spec:
  nodeName: <node_name>
  storageClassName: metalk8s
  lvmLogicalVolume:
    vgName: <vg_name>  # Choose an existing LVM VolumeGroup
    size: 10Gi  # Alertmanager LogicalVolume should have at least 1GiB capacity
  template:
    metadata:
      labels:
        app.kubernetes.io/name: 'prometheus-operator-alertmanager'
---
apiVersion: storage.metalk8s.scality.com/v1alpha1
kind: Volume
metadata:
  name: <node_name>-loki
spec:
  nodeName: <node_name>
  storageClassName: metalk8s
  lvmLogicalVolume:
    vgName: <vg_name>  # Choose an existing LVM VolumeGroup
    size: 10Gi  # Loki LogicalVolume should have at least 10GiB capacity
  template:
    metadata:
      labels:
        app.kubernetes.io/name: 'loki'
```

Create Volumes objects

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

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

For more details on the available options for storage management, see *this section of the Operational Guide*.

Loki volume sizing

Since the storage needs for logs greatly depends on the workload and the type of application that run on top of the MetalK8s cluster, you need to refer to the documentation provided by your applications to define the ideal size for the volume.

We still provide some hints for the worst case, which is very unlikely. If the entropy of log messages is high, which makes them almost incompressible, you will need around **12Mb** per thousands of event per hour for an average log line of **512 bytes**.

For **60000** events per hour, with the default retention of **2 weeks**:

```
60 (1000 events) * 24 (hours per day) * 7 (days per week) * 3 (weeks) * 12 Mb =~ 355 Gb
```

This formula is given to calculate the worst case scenario, but with real application logs, it should be drastically lower.

Regarding the MetalK8s cluster itself (internal services and system logs), **1Gb** per week of retention should be sufficient in most cases.

> **Warning:** When you calculate the storage needs, you must always add an extra week to your actual retention, because of the current week of logs.
>
> Since there is no size-based purge mechanism, it is also recommended to add a security margin of +50% volume space, in case of log burst.
>
> Also, when creating the volume, you should take into account the potential growth of the cluster and workload.

1.6.2 Changing credentials

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

To change Grafana or MetalK8s GUI user credentials, follow *this procedure*.

1.6.3 Validating the deployment

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

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

   ```
   root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
                     get nodes
   NAME         STATUS   ROLES                          AGE   VERSION
   bootstrap    Ready    bootstrap,etcd,infra,master    42m   v1.15.5
   node-1       Ready    etcd,infra,master              26m   v1.15.5
   node-2       Ready    etcd,infra,master              25m   v1.15.5
   ```

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

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

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

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

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
                  get pods --namespace kube-system
NAME                                    READY   STATUS    RESTARTS   AGE
apiserver-proxy-bootstrap               1/1     Running   0          43m
apiserver-proxy-node-1                  1/1     Running   0          2m28s
apiserver-proxy-node-2                  1/1     Running   0          9m
calico-kube-controllers-6d8db9bcf5-w5w94   1/1     Running   0          43m
calico-node-4vxpp                       1/1     Running   0          43m
calico-node-hvlkx                       1/1     Running   7          23m
calico-node-jhj4r                       1/1     Running   0          8m59s
coredns-8576b4bf99-lfjfc                1/1     Running   0          43m
coredns-8576b4bf99-tnt6b                1/1     Running   0          43m
etcd-bootstrap                          1/1     Running   0          43m
etcd-node-1                             1/1     Running   0          3m47s
etcd-node-2                             1/1     Running   3          8m58s
kube-apiserver-bootstrap                1/1     Running   0          43m
kube-apiserver-node-1                   1/1     Running   0          2m45s
kube-apiserver-node-2                   1/1     Running   0          7m31s
kube-controller-manager-bootstrap       1/1     Running   3          44m
kube-controller-manager-node-1          1/1     Running   1          2m39s
kube-controller-manager-node-2          1/1     Running   2          7m25s
kube-proxy-gnxtp                        1/1     Running   0          28m
kube-proxy-kvtjm                        1/1     Running   0          43m
kube-proxy-vggzg                        1/1     Running   0          27m
kube-scheduler-bootstrap                1/1     Running   1          44m
kube-scheduler-node-1                   1/1     Running   0          2m39s
kube-scheduler-node-2                   1/1     Running   0          7m25s
repositories-bootstrap                  1/1     Running   0          44m
salt-master-bootstrap                   2/2     Running   0          44m
storage-operator-756b87c78f-mjqc5       1/1     Running   1          43m
```

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

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

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

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

https://<first-node-ip>:2379 is healthy: successfully committed proposal: took = 16.
↪285672ms
https://<second-node-ip>:2379 is healthy: successfully committed proposal: took =␣
↪43.462092ms
https://<third-node-ip>:2379 is healthy: successfully committed proposal: took = 52.
↪879358ms
```

4. Finally, check that the exposed services are accessible, using the information from *this document*.

1.7 Accessing Cluster Services

1.7.1 MetalK8s GUI

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

Gather Required Information

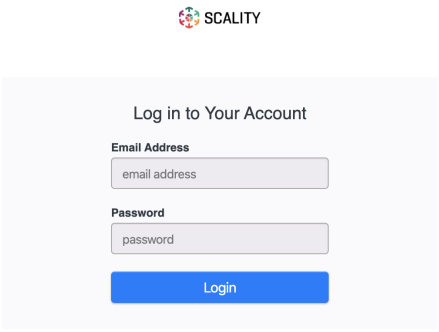
Get the ingress control plane IP.

```
root@bootstrap $ kubectl --kubeconfig=/etc/kubernetes/admin.conf \
  get svc -n metalk8s-ingress ingress-nginx-control-plane-controller \
  -o=jsonpath='{.spec.externalIPs[0]}{"\n"}'
<the ingress control plane IP>
```

Use MetalK8s UI

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

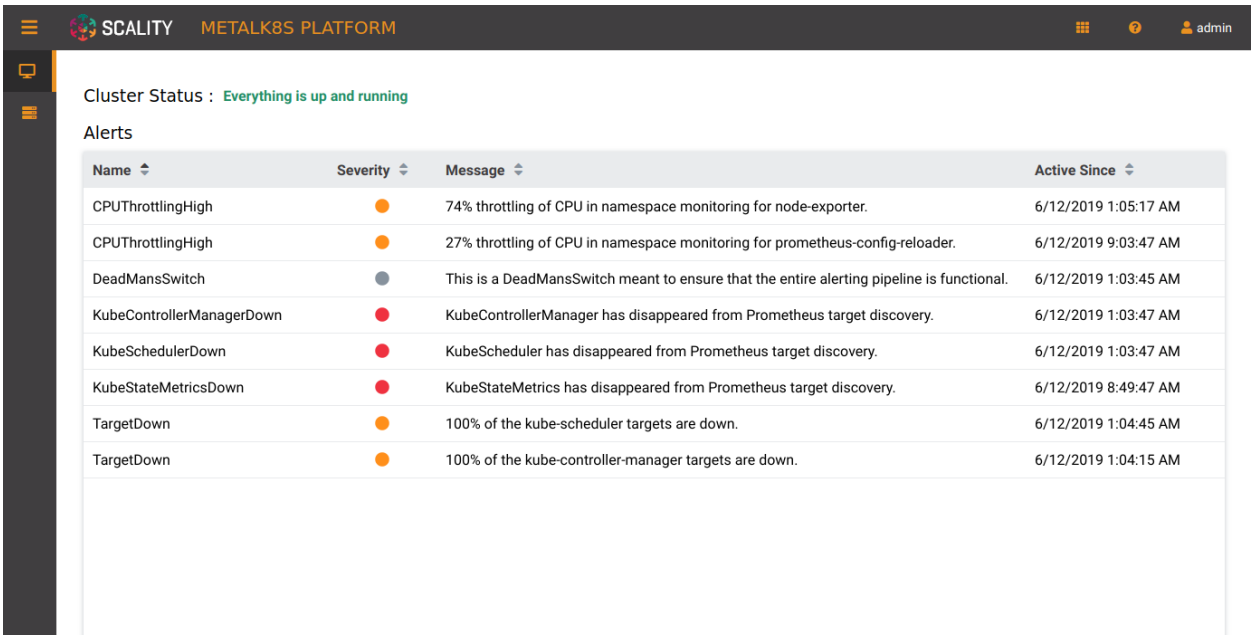
The login page is loaded, and should resemble the following:



Log in with the default login / password (`admin@metalk8s.invalid` / `password`).

> **Note:** To change the default password as provided above, refer to *this procedure*.

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*.

1.7.2 Grafana

Grafana is available on the same host as the MetalK8s UI, under `/grafana`. Log in with the default credentials: `admin@metalk8s.invalid` / `password`.

1.7.3 Salt

MetalK8s uses SaltStack to manage the cluster. The Salt Master runs in a *Pod* on the *Bootstrap node*.

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

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

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

1.8 Advanced guide

1.8.1 Multiple CIDRs network

In the Bootstrap Configuration it's possible to provide several CIDRs for a single network, it's needed when several nodes does not sit in the same network.

```
networks:
  controlPlane:
    cidr:
      - 10.100.1.0/28
      - 10.200.1.0/28
  workloadPlane:
    cidr:
      - 10.100.2.0/28
      - 10.200.2.0/28
```

This kind of deployment needs good knowledge about networking, as each workload node needs to be able to communicate with all others, even those in a different workload CIDR.

In this case *IP-in-IP encapsulation* is likely needed.

Some explanation can be found about this subject in Calico documentation.

1.9 Troubleshooting

This section describes common issues users face during and after a MetalK8s installation.

If your issue is not presented here, create a GitHub issue or open a new GitHub discussion.

1.9.1 Bootstrap Installation Errors

Bootstrap installation fails for no obvious reason

If the Metalk8s installation fails and the console output does not provide enough information to identify the cause of the failure, re-run the installation with the verbose flag (`--verbose`).

```
root@bootstrap $ /srv/scality/metalk8s-2.11.7/bootstrap.sh --verbose
```

Errors after restarting the bootstrap node

If you reboot the bootstrap node and some containers (especially the salt-master container) do not start, perform the following checks:

1. Ensure that the MetalK8s ISO is mounted properly.

   ```
   [root@bootstrap vagrant]# mount | grep /srv/scality/metalk8s-2.11.7
   /home/centos/metalk8s.iso on /srv/scality/metalk8s-2.11.7 type iso9660 (ro,relatime)
   ```

2. If the ISO is unmounted, run the following command to check the the status of the ISO file and remount it automatically.

```
[root@bootstrap vagrant]# salt-call state.sls metalk8s.archives.mounted␣
↪saltenv=metalk8s-2.11.7
 Summary for local
 ------------
 Succeeded: 3
 Failed:    0
```

Bootstrap fails and console log is unscrollable

If the bootstrap process fails during MetalK8s installation and the console output is unscrollable, consult the bootstrap logs in `/var/log/metalk8s/bootstrap.log`.

Bootstrap fails with "Invalid networks:service CIDR - no route exists"

This error happens if there is no route matching this network CIDR and no default route configured.

You can solve this issue by either adding a default route to your host or adding a dummy network interface used to define a route for this network.

Configuring a default route:

To configure a default route, refer to the official documentation of your Linux distribution.

Configuring a dummy interface:

CentOS / RHEL 7

Create the `dummy-metalk8s` interface configuration:

```
cat > /etc/sysconfig/network-scripts/ifcfg-dummy-metalk8s << 'EOF'
ONBOOT=yes
DEVICE=dummy
NM_CONTROLLED=no
NAME=dummy-metalk8s
EOF
```

Create the `ifup-dummy` network script:

```
cat > /etc/sysconfig/network-scripts/ifup-dummy << 'EOF'
#!/bin/sh
# Network configuration file for dummy network interface

. /etc/init.d/functions

cd /etc/sysconfig/network-scripts
. ./network-functions

[ -f ../network ] && . ../network

CONFIG=${1}
```

(continues on next page)

(continued from previous page)

```
need_config "${CONFIG}"
source_config

modprobe --first-time ${DEVICETYPE} numdummies=0 2> /dev/null || echo dummy␣
↪module already loaded
ip link add ${DEVNAME} type ${DEVICETYPE}
[[ -n "${IPADDR}" && -n "${NETMASK}" ]] && ip address add ${IPADDR}/${NETMASK}␣
↪dev ${DEVNAME}
ip link set ${DEVNAME} up
/etc/sysconfig/network-scripts/ifup-routes ${DEVICE} ${NAME}
EOF

chmod +x /etc/sysconfig/network-scripts/ifup-dummy
```

Create the `ifdown-dummy` network script:

```
cat > /etc/sysconfig/network-scripts/ifdown-dummy << 'EOF'
#!/bin/sh
. /etc/init.d/functions

cd /etc/sysconfig/network-scripts
. ./network-functions

[ -f ../network ] && . ../network

CONFIG=${1}

need_config "${CONFIG}"

source_config

ip link set ${DEVNAME} down
ip link del ${DEVNAME} type ${DEVICETYPE}
EOF

chmod +x /etc/sysconfig/network-scripts/ifdown-dummy
```

Create the `route-dummy-metalk8s` network script:

```
cat > /etc/sysconfig/network-scripts/route-dummy-metalk8s << EOF
$(salt-call --local pillar.get networks:service --out=txt | cut -d' ' -f2-)␣
↪dev dummy-metalk8s
EOF
```

Start the `dummy-metalk8s` interface:

```
ifup dummy-metalk8s
```

CentOS / RHEL 8 (and other NetworkManager based dists)

Retrieve the service network CIDR:

```
salt-call --local pillar.get networks:service --out=txt | cut -d' ' -f2-
```

Create the dummy-metalk8s interface:

```
nmcli connection add type dummy ifname dummy-metalk8s ipv4.method manual ipv4.
↪addresses <dummy-iface-ip> ipv4.routes <network-cidr>
```

Note: Replace <dummy-iface-ip> by any available IP in the previously retrieved network CIDR (e.g. 10.96.10.96 for a 10.96.0.0/12 network CIDR) and <network-cidr> by the network CIDR.

Start the dummy-metalk8s interface:

```
nmcli connection up dummy-dummy-metalk8s
```

1.9.2 Pod and Service CIDR Conflicts

If, after installing a MetalK8s cluster you notice routing issues in pod-to-pod communication:

1. Check the configured values for the internal pod and service networks.

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

2. Ensure that the configured IP ranges (CIDR notation) do not conflict with your infrastructure.

CHAPTER

TWO

OPERATION

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

2.1 Cluster Monitoring

This section covers the MetalK8s monitoring and alerting stack operations. It also describes the metrics monitored using Prometheus, with the list of pre-configured alerting and recording rules.

2.1.1 Monitoring Stack

MetalK8s ships with a monitoring stack that uses charts, counts, and graphs to provide a cluster-wide view of cluster health, pod status, node status, and network traffic status. Access the *Grafana Service* for monitored statistics provided once MetalK8s has been deployed.

The MetalK8s monitoring stack consists of the following main components:

- *Alertmanager*
- *Grafana*
- *Kube-state-metrics*
- *Prometheus*
- *Prometheus Node-exporter*

2.1.2 Prometheus

In a MetalK8s cluster, the Prometheus service records real-time metrics in a time series database. Prometheus can query a list of data sources called "exporters" at a specific polling frequency, and aggregate this data across the various sources.

Prometheus uses a special language, Prometheus Query Language (PromQL), to write alerting and recording rules.

Default Alert Rules

Alert rules enable a user to specify a condition that must occur before an external system like Slack is notified. For example, a MetalK8s administrator might want to raise an alert for any node that is unreachable for more than one minute.

Out of the box, MetalK8s ships with preconfigured alert rules, which are written as PromQL queries. The table below outlines all the preconfigured alert rules exposed from a newly deployed MetalK8s cluster.

To customize predefined alert rules, refer to *Prometheus Configuration Customization*.

Table 1: Default Prometheus Alerting rules

Name	Severity	Description
AlertingServiceAtRisk	critical	The alerting service is at risk.
ClusterAtRisk	critical	The cluster is at risk.
CoreServicesAtRisk	critical	The Core services are at risk.
KubernetesControlPlaneAtRisk	critical	The Kubernetes control plane is at risk.
MonitoringServiceAtRisk	critical	The monitoring service is at risk.
NodeAtRisk	critical	The node {{ $labels.instance }} is at risk.
ObservabilityServicesAtRisk	critical	The observability services are at risk.
PlatformServicesAtRisk	critical	The Platform services are at risk.
SystemPartitionAtRisk	critical	The system partition {{ $labels.mountpoint }} on node {{ $labe
VolumeAtRisk	critical	The volume {{ $labels.persistentvolumeclaim }} in namespace {
AccessServicesDegraded	warning	The Access services are degraded.
AlertingServiceDegraded	warning	The alerting service is degraded.
AuthenticationServiceDegraded	warning	The Authentication service for K8S API is degraded.
BootstrapServicesDegraded	warning	The MetalK8s Bootstrap services are degraded.
ClusterDegraded	warning	The cluster is degraded.
CoreServicesDegraded	warning	The Core services are degraded.
DashboardingServiceDegraded	warning	The dashboarding service is degraded.
IngressControllerServicesDegraded	warning	The Ingress Controllers for control plane and workload plane are
KubernetesControlPlaneDegraded	warning	The Kubernetes control plane is degraded.
LoggingServiceDegraded	warning	The logging service is degraded.
MonitoringServiceDegraded	warning	The monitoring service is degraded.
NetworkDegraded	warning	The network is degraded.
NodeDegraded	warning	The node {{ $labels.instance }} is degraded.
ObservabilityServicesDegraded	warning	The observability services are degraded.
PlatformServicesDegraded	warning	The Platform services are degraded.
SystemPartitionDegraded	warning	The system partition {{ $labels.mountpoint }} on node {{ $labe
VolumeDegraded	warning	The volume {{ $labels.persistentvolumeclaim }} in namespace {
AlertmanagerClusterCrashlooping	critical	Half or more of the Alertmanager instances within the same clus
AlertmanagerClusterDown	critical	Half or more of the Alertmanager instances within the same clus
AlertmanagerClusterFailedToSendAlerts	critical	All Alertmanager instances in a cluster failed to send notificatior
AlertmanagerClusterFailedToSendAlerts	warning	All Alertmanager instances in a cluster failed to send notificatior
AlertmanagerConfigInconsistent	critical	Alertmanager instances within the same cluster have different co
AlertmanagerFailedReload	critical	Reloading an Alertmanager configuration has failed.
AlertmanagerFailedToSendAlerts	warning	An Alertmanager instance failed to send notifications.
AlertmanagerMembersInconsistent	critical	A member of an Alertmanager cluster has not found all other clu
etcdGRPCRequestsSlow	critical	etcd cluster "{{ $labels.job }}": gRPC requests to {{ $labels.grp
etcdHTTPRequestsSlow	warning	etcd instance {{ $labels.instance }} HTTP requests to {{ $labels
etcdHighCommitDurations	warning	etcd cluster "{{ $labels.job }}": 99th percentile commit duration
etcdHighFsyncDurations	warning	etcd cluster "{{ $labels.job }}": 99th percentile fync durations a

Table 1 – continued from previous page

Name	Severity	Description
etcdHighNumberOfFailedGRPCRequests	critical	etcd cluster "{{ $labels.job }}": {{ $value }}% of requests for {
etcdHighNumberOfFailedGRPCRequests	warning	etcd cluster "{{ $labels.job }}": {{ $value }}% of requests for {
etcdHighNumberOfFailedHTTPRequests	critical	{{ $value }}% of requests for {{ $labels.method }} failed on etc
etcdHighNumberOfFailedHTTPRequests	warning	{{ $value }}% of requests for {{ $labels.method }} failed on etc
etcdHighNumberOfFailedProposals	warning	etcd cluster "{{ $labels.job }}": {{ $value }} proposal failures w
etcdHighNumberOfLeaderChanges	warning	etcd cluster "{{ $labels.job }}": instance {{ $labels.instance }}
etcdInsufficientMembers	critical	etcd cluster "{{ $labels.job }}": insufficient members ({{ $value
etcdMemberCommunicationSlow	warning	etcd cluster "{{ $labels.job }}": member communication with {
etcdNoLeader	critical	etcd cluster "{{ $labels.job }}": member {{ $labels.instance }}
TargetDown	warning	One or more targets are unreachable.
Watchdog	none	An alert that should always be firing to certify that Alertmanager
KubeAPIErrorBudgetBurn	warning	The API server is burning too much error budget.
KubeAPIErrorBudgetBurn	critical	The API server is burning too much error budget.
KubeAPIErrorBudgetBurn	warning	The API server is burning too much error budget.
KubeAPIErrorBudgetBurn	critical	The API server is burning too much error budget.
KubeStateMetricsListErrors	critical	kube-state-metrics is experiencing errors in list operations.
KubeStateMetricsShardingMismatch	critical	kube-state-metrics sharding is misconfigured.
KubeStateMetricsShardsMissing	critical	kube-state-metrics shards are missing.
KubeStateMetricsWatchErrors	critical	kube-state-metrics is experiencing errors in watch operations.
KubeContainerWaiting	warning	Pod container waiting longer than 1 hour
KubeDaemonSetMisScheduled	warning	DaemonSet pods are misscheduled.
KubeDaemonSetNotScheduled	warning	DaemonSet pods are not scheduled.
KubeDaemonSetRolloutStuck	warning	DaemonSet rollout is stuck.
KubeDeploymentGenerationMismatch	warning	Deployment generation mismatch due to possible roll-back
KubeDeploymentReplicasMismatch	warning	Deployment has not matched the expected number of replicas.
KubeHpaMaxedOut	warning	HPA is running at max replicas
KubeHpaReplicasMismatch	warning	HPA has not matched descired number of replicas.
KubeJobCompletion	warning	Job did not complete in time
KubeJobFailed	warning	Job failed to complete.
KubePodCrashLooping	warning	Pod is crash looping.
KubePodNotReady	warning	Pod has been in a non-ready state for more than 15 minutes.
KubeStatefulSetGenerationMismatch	warning	StatefulSet generation mismatch due to possible roll-back
KubeStatefulSetReplicasMismatch	warning	Deployment has not matched the expected number of replicas.
KubeStatefulSetUpdateNotRolledOut	warning	StatefulSet update has not been rolled out.
CPUThrottlingHigh	info	Processes experience elevated CPU throttling.
KubeCPUOvercommit	warning	Cluster has overcommitted CPU resource requests.
KubeCPUQuotaOvercommit	warning	Cluster has overcommitted CPU resource requests.
KubeMemoryOvercommit	warning	Cluster has overcommitted memory resource requests.
KubeMemoryQuotaOvercommit	warning	Cluster has overcommitted memory resource requests.
KubeQuotaAlmostFull	info	Namespace quota is going to be full.
KubeQuotaExceeded	warning	Namespace quota has exceeded the limits.
KubeQuotaFullyUsed	info	Namespace quota is fully used.
KubePersistentVolumeErrors	critical	PersistentVolume is having issues with provisioning.
KubePersistentVolumeFillingUp	critical	PersistentVolume is filling up.
KubePersistentVolumeFillingUp	warning	PersistentVolume is filling up.
KubeClientErrors	warning	Kubernetes API server client is experiencing errors.
KubeVersionMismatch	warning	Different semantic versions of Kubernetes components running.
AggregatedAPIDown	warning	An aggregated API is down.
AggregatedAPIErrors	warning	An aggregated API has reported errors.

Table 1 – continued from previous page

Name	Severity	Description
KubeAPIDown	critical	Target disappeared from Prometheus target discovery.
KubeAPITerminatedRequests	warning	The apiserver has terminated {{ $value | humanizePercentage }}
KubeClientCertificateExpiration	critical	Client certificate is about to expire.
KubeClientCertificateExpiration	warning	Client certificate is about to expire.
KubeControllerManagerDown	critical	Target disappeared from Prometheus target discovery.
KubeNodeNotReady	warning	Node is not ready.
KubeNodeReadinessFlapping	warning	Node readiness status is flapping.
KubeNodeUnreachable	warning	Node is unreachable.
KubeletClientCertificateExpiration	critical	Kubelet client certificate is about to expire.
KubeletClientCertificateExpiration	warning	Kubelet client certificate is about to expire.
KubeletClientCertificateRenewalErrors	warning	Kubelet has failed to renew its client certificate.
KubeletDown	critical	Target disappeared from Prometheus target discovery.
KubeletPlegDurationHigh	warning	Kubelet Pod Lifecycle Event Generator is taking too long to relis
KubeletPodStartUpLatencyHigh	warning	Kubelet Pod startup latency is too high.
KubeletServerCertificateExpiration	critical	Kubelet server certificate is about to expire.
KubeletServerCertificateExpiration	warning	Kubelet server certificate is about to expire.
KubeletServerCertificateRenewalErrors	warning	Kubelet has failed to renew its server certificate.
KubeletTooManyPods	warning	Kubelet is running at capacity.
KubeSchedulerDown	critical	Target disappeared from Prometheus target discovery.
NodeClockNotSynchronising	warning	Clock on {{ $labels.instance }} is not synchronising. Ensure NT
NodeClockSkewDetected	warning	Clock on {{ $labels.instance }} is out of sync by more than 300s
NodeFileDescriptorLimit	critical	Kernel is predicted to exhaust file descriptors limit soon.
NodeFileDescriptorLimit	warning	Kernel is predicted to exhaust file descriptors limit soon.
NodeFilesystemAlmostOutOfFiles	critical	Filesystem has less than 8% inodes left.
NodeFilesystemAlmostOutOfFiles	warning	Filesystem has less than 15% inodes left.
NodeFilesystemAlmostOutOfSpace	critical	Filesystem has less than 12% space left.
NodeFilesystemAlmostOutOfSpace	warning	Filesystem has less than 20% space left.
NodeFilesystemFilesFillingUp	critical	Filesystem is predicted to run out of inodes within the next 4 hou
NodeFilesystemFilesFillingUp	warning	Filesystem is predicted to run out of inodes within the next 24 ho
NodeFilesystemSpaceFillingUp	critical	Filesystem is predicted to run out of space within the next 4 hour
NodeFilesystemSpaceFillingUp	warning	Filesystem is predicted to run out of space within the next 24 hou
NodeHighNumberConntrackEntriesUsed	warning	Number of conntrack are getting close to the limit
NodeNetworkReceiveErrs	warning	Network interface is reporting many receive errors.
NodeNetworkTransmitErrs	warning	Network interface is reporting many transmit errors.
NodeRAIDDegraded	critical	RAID Array is degraded
NodeRAIDDiskFailure	warning	Failed device in RAID array
NodeTextFileCollectorScrapeError	warning	Node Exporter text file collector failed to scrape.
NodeNetworkInterfaceFlapping	warning	Network interface is often changin it's status
PrometheusBadConfig	critical	Failed Prometheus configuration reload.
PrometheusDuplicateTimestamps	warning	Prometheus is dropping samples with duplicate timestamps.
PrometheusErrorSendingAlertsToAnyAlertmanager	critical	Prometheus encounters more than 3% errors sending alerts to an
PrometheusErrorSendingAlertsToSomeAlertmanagers	warning	Prometheus has encountered more than 1% errors sending alerts
PrometheusLabelLimitHit	warning	Prometheus has dropped targets because some scrape configs ha
PrometheusMissingRuleEvaluations	warning	Prometheus is missing rule evaluations due to slow rule group ev
PrometheusNotConnectedToAlertmanagers	warning	Prometheus is not connected to any Alertmanagers.
PrometheusNotIngestingSamples	warning	Prometheus is not ingesting samples.
PrometheusNotificationQueueRunningFull	warning	Prometheus alert notification queue predicted to run full in less t
PrometheusOutOfOrderTimestamps	warning	Prometheus drops samples with out-of-order timestamps.
PrometheusRemoteStorageFailures	critical	Prometheus fails to send samples to remote storage.

Table 1 – continued from previous page

Name	Severity	Description
PrometheusRemoteWriteBehind	critical	Prometheus remote write is behind.
PrometheusRemoteWriteDesiredShards	warning	Prometheus remote write desired shards calculation wants to run
PrometheusRuleFailures	critical	Prometheus is failing rule evaluations.
PrometheusTSDBCompactionsFailing	warning	Prometheus has issues compacting blocks.
PrometheusTSDBReloadsFailing	warning	Prometheus has issues reloading blocks from disk.
PrometheusTargetLimitHit	warning	Prometheus has dropped targets because some scrape configs ha
PrometheusTargetSyncFailure	critical	Prometheus has failed to sync targets.
PrometheusOperatorListErrors	warning	Errors while performing list operations in controller.
PrometheusOperatorNodeLookupErrors	warning	Errors while reconciling Prometheus.
PrometheusOperatorNotReady	warning	Prometheus operator not ready
PrometheusOperatorReconcileErrors	warning	Errors while reconciling controller.
PrometheusOperatorRejectedResources	warning	Resources rejected by Prometheus operator
PrometheusOperatorSyncFailed	warning	Last controller reconciliation failed
PrometheusOperatorWatchErrors	warning	Errors while performing watch operations in controller.

Snapshot Prometheus Database

To snapshot the database, you must first *enable the Prometheus admin API*.

To generate a snapshot, use the *sosreport utility* with the following options:

```
root@host # sosreport --batch --build -o metalk8s -kmetalk8s.prometheus-snapshot=True
```

The name of the generated archive is printed on the console output and the Prometheus snapshot can be found under `prometheus_snapshot` directory.

Warning: You must ensure you have sufficient disk space (at least the size of the Prometheus volume) under `/var/tmp` or change the archive destination with `--tmp-dir=<new_dest>` option.

2.2 Account Administration

This section covers MetalK8s account administration operations, from user authentication and identity management to user authorization.

2.2.1 User Authentication and Identity Management

In MetalK8s, user authentication and identity management are driven by the integration of `kube-apiserver` and Dex, an OpenID Connect (OIDC) provider.

Kubernetes API enables OIDC as one authentication strategy (it also supports certificate-based authentication) by trusting Dex as an OIDC provider.

Dex can authenticate users against:

- a static user store (stored in configuration),
- a connector-based interface, allowing plug-ins from such external providers as LDAP, SAML, GitHub, Active Directory and others to plug in.

Note: Out of the box, MetalK8s enables OIDC-based authentication for its UI and Grafana service.

Administering Grafana and MetalK8s UI

When MetalK8s is first installed, the UI and Grafana service are set with the default login credentials `admin@metalk8s.invalid`, and `password`.

This default user is defined as a static user in the Dex configuration to enable MetalK8s administrators' first access to these services. Change the default password after the first login.

Note: The MetalK8s UI and Grafana are both configured to use OIDC as an authentication mechanism, and trust Dex as a provider. Changing the Dex configuration, including the default credentials, affects both UIs.

To access the MetalK8s UI and Grafana service, refer to *Accessing Cluster Services*.

Adding a Static User

To add a static user for the MetalK8s UI and or the Grafana service, perform the following steps from the bootstrap node.

1. Generate a bcrypt hash of your password.

```
root@bootstrap $ htpasswd -nBC 14 "" | tr -d ':'
New password:
Re-type new password:
<your hash here, starting with "$2y$14$">
```

2. Generate a unique identifier.

```
root@bootstrap $ python -c 'import uuid; print uuid.uuid4()'
```

3. Add a new entry in the `staticPasswords` list. Use the password hash and user ID previously generated, and choose a new email and user name.

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
                    edit configmap metalk8s-dex-config -n metalk8s-auth
```

```
# [...]
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com/v1alpha2
    kind: DexConfiguration
    spec:
      # [...]
      config:
        # [...]
        staticPasswords:
          # [...]
          - email: "<email>"
            hash: "<generated-password-hash>"
```

(continues on next page)

(continued from previous page)

```
            username: "<username>"
            userID: "<generated-identifier>"
```

4. Apply your changes.

```
root@bootstrap $ STATES=$(printf ",metalk8s.addons.%s.deployed" \
                           dex prometheus-operator ui)
root@bootstrap $ kubectl exec -n kube-system -c salt-master \
                  --kubeconfig /etc/kubernetes/admin.conf \
                  salt-master-bootstrap -- salt-run state.sls \
                  "${STATES:1}" saltenv=metalk8s-2.11.7
```

5. Bind the user to an existing (Cluster) Role using *a ClusterRoleBlinding*.

6. Check that the user has been successfully added. If so, log into the MetalK8s UI using the new email and password.

Changing Static User Password

Important: Default admin user

A new MetalK8s installation is supplied with a default administrator account and a predefined password (see *Use MetalK8s UI*). Change this password if the control plane network is accessible to untrusted clients.

To change the default password for the MetalK8s UI or the Grafana service, perform the following steps from the Bootstrap node.

1. Generate a bcrypt hash of the new password.

```
root@bootstrap $ htpasswd -nBC 14 "" | tr -d ':'
New password:
Re-type new password:
<your hash here, starting with "$2y$14$">
```

2. Find the entry for the selected user in the `staticPasswords` list and update its hash.

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
                   edit configmap metalk8s-dex-config -n metalk8s-auth
```

```
# [...]
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com/v1alpha2
    kind: DexConfiguration
    spec:
      # [...]
      config:
        # [...]
        staticPasswords:
          # [...]
          - email: "<previous-email>"
            hash: "<new-password-hash>"
```

(continues on next page)

(continued from previous page)

```
            username: "<previous-username>"
            userID: "<previous-identifier>"
          # [...]
```

3. Apply your changes.

```
root@bootstrap $ kubectl exec -n kube-system -c salt-master \
                    --kubeconfig /etc/kubernetes/admin.conf \
                    salt-master-bootstrap -- salt-run state.sls \
                    metalk8s.addons.dex.deployed saltenv=metalk8s-2.11.7
```

4. Check that the password has been changed. If so, log into the MetalK8s UI using the new password.

2.2.2 User Authorization

Kubernetes API

To authorize users and groups against the Kubernetes API, the *API Server* relies on RBAC (Role-Based Access Control), through the use of special API objects:

- **Roles** and **ClusterRoles**, which define specific permissions on a set of API resources,
- **RoleBindings** and **ClusterRoleBindings**, which map a user or group to a set of Roles or ClusterRoles.

Note: MetalK8s includes pre-provisioned ClusterRoles. Platform administrators can create new Roles or ClusterRoles or refer to existing ones.

ClusterRoles

- Obtain the list of available ClusterRoles.

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

- Describe a ClusterRole for more information.

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
                    describe clusterrole <name>
```

- The pre-provisioned static user **admin@metalk8s.invalid** is already bound to the **cluster-admin** ClusterRole, which grants cluster-wide permissions to all exposed APIs.

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
                    describe clusterrole cluster-admin
Name:         cluster-admin
Labels:       kubernetes.io/bootstrapping=rbac-defaults
Annotations:  rbac.authorization.kubernetes.io/autoupdate: true
PolicyRule:
  Resources  Non-Resource URLs  Resource Names  Verbs
  ---------  -----------------  --------------  -----
```

(continues on next page)

(continued from previous page)

```
  *.*        []                []              [*]
             [*]               []              [*]
```

For more information on Kubernetes authorization mechanisms, refer to the RBAC documentation.

ClusterRoleBindings

To bind one or more users to an existing ClusterRole in all namespaces, follow this procedure.

1. Create a ClusterRoleBinding manifest (`role_binding.yaml`) from the following template.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: <role-binding-name-of-your-choice>
subjects:
  - kind: User
    name: <email>
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: <target-cluster-role>
  apiGroup: rbac.authorization.k8s.io
```

2. Apply the manifest.

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
                    apply -f role_binding.yaml
```

To bind one or more groups to an existing ClusterRole in all namespaces, follow this procedure.

1. Create a ClusterRoleBinding manifest (`role_binding.yaml`) from the following template.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: <role-binding-name-of-your-choice>
subjects:
  - kind: Group
    name: <group-name>
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: <target-cluster-role>
  apiGroup: rbac.authorization.k8s.io
```

2. Apply the manifest.

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
                    apply -f role_binding.yaml
```

2.3 Cluster and Services Configurations

This section contains information describing the list of available Cluster and Services Configurations including procedures for customizing and applying any given Cluster and Services Configurations.

2.3.1 Default Service Configurations

MetalK8s addons (Alertmanager, Dex, Grafana, Prometheus and UI) ships with default runtime service configurations required for basic service deployment. Find below an exhaustive list of available default Service Configurations deployed in a MetalK8s cluster.

Alertmanager Default Configuration

Alertmanager handles alerts sent by Prometheus. It takes care of deduplicating, grouping, and routing them to the correct receiver integration such as email, PagerDuty, or OpsGenie. It also takes care of silencing and inhibition of alerts.

The default configuration values for Alertmanager are specified below:

```
# Configuration of the Alertmanager service
apiVersion: addons.metalk8s.scality.com
kind: AlertmanagerConfig
spec:
  # Configure the Alertmanager Deployment
  deployment:
    replicas: 1
  notification:
    config:
      global:
        resolve_timeout: 5m
      templates: []
      route:
        group_by: ['job']
        group_wait: 30s
        group_interval: 5m
        repeat_interval: 12h
        receiver: 'metalk8s-alert-logger'
        routes:
        - receiver: 'metalk8s-alert-logger'
          continue: True
      receivers:
        - name: 'metalk8s-alert-logger'
          webhook_configs:
            - send_resolved: True
              url: 'http://metalk8s-alert-logger:19094/'
      inhibit_rules: []
```

See *Alertmanager Configuration Customization* to override these defaults.

Dex Default Configuration

Dex is an Identity Provider that drives user authentication and identity management in a MetalK8s cluster.

The default configuration values for Dex are specified below:

```
# Defaults for configuration of Dex (OIDC)
apiVersion: addons.metalk8s.scality.com/v1alpha2
kind: DexConfig
spec:
  # Deployment configuration
  deployment:
    replicas: 2
    affinity:
      podAntiAffinity:
        soft:
          - topologyKey: kubernetes.io/hostname
        #   - topologyKey: my.second.important/topologyKey
        #     weight: 42
        # hard:
        #   - topologyKey: kubernetes.io/hostname

  # Dex server configuration
  config:
    issuer: {{ control_plane_ingress_ep }}/oidc

    storage:
      config:
        inCluster: true
      type: kubernetes

    logger:
      level: debug
      https: 0.0.0.0:5554
      tlsCert: /etc/dex/tls/https/server/tls.crt
      tlsKey: /etc/dex/tls/https/server/tls.key

    frontend:
      dir: /srv/dex/web/
      theme: scality
      issuer: MetalK8s

    connectors: []

    oauth2:
      alwaysShowLoginScreen: true
      skipApprovalScreen: true
      responseTypes: ["code", "token", "id_token"]

    expiry:
      signingKeys: "6h"
      idTokens: "24h"

    {#- FIXME: client secrets shouldn't be hardcoded #}
```

(continues on next page)

(continued from previous page)

```
    {#- TODO: allow overriding these predefined clients #}
    staticClients:
    - id: oidc-auth-client
      name: oidc-auth-client
      redirectURIs:
      - urn:ietf:wg:oauth:2.0:oob
      secret: lkfa9jaf3kfakqyeoikfjakf93k2l
      trustedPeers:
      - metalk8s-ui
      - grafana-ui
    - id: metalk8s-ui
      name: MetalK8s UI
      redirectURIs:
      - {{ control_plane_ingress_ep }}/{{ metalk8s_ui_config.spec.basePath.lstrip('/') }}
      secret: ybrMJpVMQxsiZw26MhJzCjA2ut
    - id: grafana-ui
      name: Grafana UI
      redirectURIs:
      - {{ control_plane_ingress_ep }}/grafana/login/generic_oauth
      secret: 4lqK98NcsWG5qBRHJUqYM1

    enablePasswordDB: true
    staticPasswords: []
```

See *Dex Configuration Customization* for Dex configuration customizations.

Grafana Default Configuration

Grafana is a web interface used to visualize and analyze metrics scraped by Prometheus, with nice graphs.

The default configuration values for Grafana are specified below:

```
# Configuration of the Grafana service
apiVersion: addons.metalk8s.scality.com
kind: GrafanaConfig
spec:
  # Configure the Grafana Deployment
  deployment:
    replicas: 1
```

Prometheus Default Configuration

Prometheus is responsible for monitoring all the applications and systems in the MetalK8s cluster. It scrapes and stores various metrics from these systems and then analyze them against a set of alerting rules. If a rule matches, Prometheus sends an alert to Alertmanager.

The default configuration values for Prometheus are specified below:

```
# Configuration of the Prometheus service
apiVersion: addons.metalk8s.scality.com
kind: PrometheusConfig
```

(continues on next page)

(continued from previous page)

```yaml
spec:
  # Configure the Prometheus Deployment
  deployment:
    replicas: 1
  config:
    retention_time: "10d"
    retention_size: "0"  # "0" to disable size-based retention
    enable_admin_api: false
    serviceMonitor:
      kubelet:
        scrapeTimeout: 10s
  rules:
    node_exporter:
      node_filesystem_space_filling_up:
        warning:
          hours: 24  # Hours before there is no space left
          threshold: 40  # Min space left to trigger prediction
        critical:
          hours: 4
          threshold: 20
      node_filesystem_almost_out_of_space:
        warning:
          available: 20  # Percentage of free space left
        critical:
          available: 12
      node_filesystem_files_filling_up:
        warning:
          hours: 24  # Hours before there is no inode left
          threshold: 40  # Min space left to trigger prediction
        critical:
          hours: 4
          threshold: 20
      node_filesystem_almost_out_of_files:
        warning:
          available: 15  # Percentage of free inodes left
        critical:
          available: 8
      node_network_receive_errors:
        warning:
          error_rate: 0.01  # Rate of receive errors for the last 2m
      node_network_transmit_errors:
        warning:
          error_rate: 0.01  # Rate of transmit errors for the last 2m
      node_high_number_conntrack_entries_used:
        warning:
          threshold: 0.75
      node_clock_skew_detected:
        warning:
          threshold:
            high: 0.05
            low: -0.05
      node_clock_not_synchronising:
```

(continues on next page)

(continued from previous page)

```
        warning:
          threshold: 0
      node_raid_degraded:
        critical:
          threshold: 1
      node_raid_disk_failure:
        warning:
          threshold: 1
```

Loki Default Configuration

Loki is a log aggregation system, its job is to receive logs from collectors (fluent-bit), store them on persistent storage, then make them queryable through its API.

The default configuration values for Loki are specified below:

```
# Configuration of the Loki service
apiVersion: addons.metalk8s.scality.com
kind: LokiConfig
spec:
  deployment:
    replicas: 1
    resources:
      requests:
        memory: "256Mi"
  config:
    auth_enabled: false
    chunk_store_config:
      max_look_back_period: 0s
    memberlist:
      abort_if_cluster_join_fails: false
      join_members:
        - loki-headless:7946
      dead_node_reclaim_time: 30s
      gossip_to_dead_nodes_time: 15s
      left_ingesters_timeout: 30s
      bind_addr: ["0.0.0.0"]
      bind_port: 7946
    ingester:
      chunk_block_size: 262144
      chunk_idle_period: 3m
      chunk_retain_period: 1m
      lifecycler:
        ring:
          kvstore:
            store: memberlist
      max_transfer_retries: 0
    limits_config:
      enforce_metric_name: false
      reject_old_samples: true
      reject_old_samples_max_age: 168h
```

(continues on next page)

(continued from previous page)

```
    schema_config:
      configs:
      - from: 2018-04-15
        index:
          period: 168h
          prefix: index_
        object_store: filesystem
        schema: v9
        store: boltdb
    server:
      http_listen_port: 3100
    storage_config:
      boltdb:
        directory: /data/loki/index
      filesystem:
        directory: /data/loki/chunks
    table_manager:
      retention_deletes_enabled: true
      retention_period: 336h
```

UI Default Configuration

MetalK8s UI simplifies management and monitoring of a MetalK8s cluster from a centralized user interface.

The default configuration values for MetalK8s UI are specified below:

```
# Defaults for configuration of MetalK8s UI
apiVersion: addons.metalk8s.scality.com/v1alpha2
kind: UIConfig
spec:
  auth:
    kind: "OIDC"
    providerUrl: "/oidc"
    redirectUrl: "{{ salt.metalk8s_network.get_control_plane_ingress_endpoint() }}/"
    clientId: "metalk8s-ui"
    responseType: "id_token"
    scopes: "openid profile email groups offline_access audience:server:client_id:oidc-
↪auth-client"
  title: Metalk8s Platform
  basePath: /
```

See *Metalk8s UI Configuration Customization* to override these defaults.

Shell UI Default Configuration

MetalK8s Shell UI provides a common set of features to MetalK8s UI and any other UI (both control and workload plane) configured to include the Shell UI component(s). Features exposed include: - user authentication using an OIDC provider - navigation menu items, displayed according to user groups (retrieved from OIDC)

The default Shell UI configuration values are specified below:

```
{%- set dex_defaults = salt.slsutil.renderer('salt://metalk8s/addons/dex/config/dex.yaml.
↪j2', saltenv=saltenv) %}
{%- set dex = salt.metalk8s_service_configuration.get_service_conf('metalk8s-auth',
↪'metalk8s-dex-config', dex_defaults) %}
{%- set metalk8s_ui_defaults = salt.slsutil.renderer(
        'salt://metalk8s/addons/ui/config/metalk8s-ui-config.yaml.j2', saltenv=saltenv
    )
%}

{%- set metalk8s_ui_config = salt.metalk8s_service_configuration.get_service_conf(
        'metalk8s-ui', 'metalk8s-ui-config', metalk8s_ui_defaults
    )
%}


# Defaults for shell UI configuration
apiVersion: addons.metalk8s.scality.com/v1alpha2
kind: ShellUIConfig
spec:
  oidc:
    providerUrl: "/oidc"
    redirectUrl: "{{ salt.metalk8s_network.get_control_plane_ingress_endpoint() }}/{{␣
↪metalk8s_ui_config.spec.basePath.lstrip('/') }}"
    clientId: "metalk8s-ui"
    responseType: "id_token"
    scopes: "openid profile email groups offline_access audience:server:client_id:oidc-
↪auth-client"
  userGroupsMapping:
{%- for user in dex.spec.config.staticPasswords | map(attribute='email') %}
    "{{ user }}": [metalk8s:admin]
{%- endfor %}
  discoveryUrl: "/shell/deployed-ui-apps.json"
  logo:
    light: /brand/assets/logo-light.svg
    dark: /brand/assets/logo-dark.svg
    darkRebrand: /brand/assets/logo-darkRebrand.svg
  favicon: /brand/favicon-metalk8s.svg
  canChangeLanguage: false
  canChangeTheme: false
```

See *MetalK8s Shell UI Configuration Customization* to override these defaults.

2.3.2 Service Configurations Customization

Alertmanager Configuration Customization

Default configuration for Alertmanager can be overridden by editing its Cluster and Service ConfigMap `metalk8s-alertmanager-config` in namespace `metalk8s-monitoring` under the key `data.config\.yaml`:

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
                    edit configmap -n metalk8s-monitoring \
                    metalk8s-alertmanager-config
```

The following documentation is not exhaustive and is just here to give some hints on basic usage, for more details or advanced configuration, see the official Alertmanager documentation.

Adding inhibition rule for an alert

Alert inhibition rules allow making one alert inhibit notifications for some other alerts.

For example, inhibiting alerts with a `warning` severity when there is the same alert with a `critical` severity.

```
apiVersion: v1
kind: ConfigMap
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com
    kind: AlertmanagerConfig
    spec:
      notification:
        config:
          inhibit_rules:
            - source_match:
                severity: critical
              target_match:
                severity: warning
              equal:
                - alertname
```

Adding receivers

Receivers allow configuring where the alert notifications are sent.

Here is a simple Slack receiver which makes Alertmanager send all notifications to a specific Slack channel.

```
apiVersion: v1
kind: ConfigMap
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com
    kind: AlertmanagerConfig
    spec:
      notification:
        config:
```

(continues on next page)

(continued from previous page)

```
        global:
          slack_api_url: https://hooks.slack.com/services/ABCDEFGHIJK
        route:
          receiver: slack-receiver
        receivers:
          - name: slack-receiver
            slack_configs:
              - channel: '#<your-channel>'
                send_resolved: true
```

You can find documentation here to activate incoming webhooks for your Slack workspace and retrieve the `slack_api_url` value.

Another example, with email receiver.

```
apiVersion: v1
kind: ConfigMap
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com
    kind: AlertmanagerConfig
    spec:
      notification:
        config:
          route:
            receiver: email-receiver
          receivers:
            - name: email-receiver
              email_configs:
                - to: <your-address>@<your-domain.tld>
                  from: alertmanager@<your-domain.tld>
                  smarthost: <smtp.your-domain.tld>:587
                  auth_username: alertmanager@<your-domain.tld>
                  auth_identity: alertmanager@<your-domain.tld>
                  auth_password: <password>
                  send_resolved: true
```

There are more receivers available (PagerDuty, OpsGenie, HipChat, ...).

Applying configuration

Any changes made to `metalk8s-alertmanager-config` ConfigMap must then be applied with Salt.

```
root@bootstrap $ kubectl exec --kubeconfig /etc/kubernetes/admin.conf \
                  -n kube-system -c salt-master salt-master-bootstrap -- \
                  salt-run state.sls \
                  metalk8s.addons.prometheus-operator.deployed \
                  saltenv=metalk8s-2.11.7
```

Grafana Configuration Customization

Add Extra Dashboard

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  labels:
    grafana_dashboard: '1'
  name: <grafana-dashboard-name>
  namespace: metalk8s-monitoring
data:
  <dashboard-filename>.json: |-
    <dashboard-definition>
```

Note: The ConfigMap must be deployed in *metalk8s-monitoring* namespace and the *grafana_dashboard: '1'* label in the example above is mandatory for the dashboard to be taken into account.

Then this manifest must be applied.

```
root@bootstrap $ kubectl --kubeconfig=/etc/kubernetes/admin.conf \
                  apply -f <path-to-the-manifest>
```

Prometheus Configuration Customization

Default configuration for Prometheus can be overridden by editing its Cluster and Service ConfigMap `metalk8s-prometheus-config` in namespace `metalk8s-monitoring` under the key `data.config.yaml`:

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
                  edit configmap -n metalk8s-monitoring \
                  metalk8s-prometheus-config
```

Change Retention Time

Prometheus is deployed with a retention based on time (10d). This value can be overriden:

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: metalk8s-prometheus-config
  namespace: metalk8s-monitoring
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com
    kind: PrometheusConfig
    spec:
```

(continues on next page)

(continued from previous page)

```
    config:
      retention_time: 30d
```

Note: Supported time units are y, w, d, h, m s and ms (years, weeks, days, hours, minutes, seconds and milliseconds).

Then *apply the configuration*.

Set Retention Size

Prometheus is deployed with the size-based retention disabled. This functionality can be actived:

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: metalk8s-prometheus-config
  namespace: metalk8s-monitoring
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com
    kind: PrometheusConfig
    spec:
      config:
        retention_size: 10GB
```

Note: Supported size units are B, KB, MB, GB, TB and PB.

Warning: Prometheus does not take the write-ahead log (WAL) size in account to calculate the retention, so the actual disk consumption can be greater than *retention_size*. You should at least add a 10% margin to be safe. (i.e.: set *retention_size* to 9GB for a 10GB volume)

Both size and time based retentions can be activated at the same time.

Then *apply the configuration*.

Set Kubelet metrics scrape tiemout

In some cases (e.g. when using a lot of sparse loop devices), the kubelet metrics endpoint can be very slow to answer and the Prometheus' default 10s scrape timeout may not be sufficient. To avoid timeouts and thus losing metrics, you can customize the scrape timeout as follows:

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: metalk8s-prometheus-config
```

(continues on next page)

(continued from previous page)

```
  namespace: metalk8s-monitoring
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com
    kind: PrometheusConfig
    spec:
      config:
        serviceMonitor:
          kubelet:
            scrapeTimeout: 30s
```

Then *apply the configuration*.

Predefined Alert Rules Customization

A subset of the predefined Alert rules can be customized, the exhaustive list can be found *here*.

For example, to change the threshold for the disk space alert (% of free space left) from 5% to 10%, simply do:

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: metalk8s-prometheus-config
  namespace: metalk8s-monitoring
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com
    kind: PrometheusConfig
    spec:
      rules:
        node_exporter:
          node_filesystem_almost_out_of_space:
            warning:
              available: 10
```

Then *apply the configuration*.

Enable Prometheus Admin API

For security reasons, Prometheus Admin API is disabled by default. It can be enabled with the following:

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: metalk8s-prometheus-config
  namespace: metalk8s-monitoring
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com
```

(continues on next page)

(continued from previous page)

```
    kind: PrometheusConfig
    spec:
      config:
        enable_admin_api: true
```

Then *apply the configuration*.

Adding New Rules

Alerting rules allow defining alert conditions based on `PromQL` expressions and to send notifications about these alerts to Alertmanager.

In order to add Alert rules, a new `PrometheusRule` manifest must be created.

```
---
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  labels:
    metalk8s.scality.com/monitor: ''
  name: <prometheus-rule-name>
  namespace: <namespace-name>
spec:
  groups:
  - name: <rules-group-name>
    rules:
    - alert: <alert-rule-name>
      annotations:
        description: "some description"
        summary: "alert summary"
      expr: <PromQL-expression>
      for: 1h
      labels:
        severity: warning
```

Note: The *metalk8s.scality.com/monitor: ''* label in the example above is mandatory for Prometheus to take the new rules into account.

Then this manifest must be applied.

```
root@bootstrap $ kubectl --kubeconfig=/etc/kubernetes/admin.conf \
                  apply -f <path-to-the-manifest>
```

For more details on Alert Rules, see the official Prometheus alerting rules documentation

Adding New Service to Monitor

To tell monitor to scrape metrics for a Pod, a new `ServiceMonitor` manifest must be created.

```
---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    metalk8s.scality.com/monitor: ''
  name: <service-monitor-name>
  namespace: <namespace-name>
spec:
  endpoints:
    - port: <port-name>
  namespaceSelector:
    matchNames:
      - <namespace-name>
  selector:
    matchLabels:
      app.kubernetes.io/name: <app-name>
```

Note: The *metalk8s.scality.com/monitor: ''* label in the example above is mandatory for Prometheus to take the new service to monitor into account.

Then this manifest must be applied.

```
root@bootstrap $ kubectl --kubeconfig=/etc/kubernetes/admin.conf \
                    apply -f <path-to-the-manifest>
```

For details and an example, see the Prometheus Operator documentation.

Applying configuration

Any changes made to `metalk8s-prometheus-config` ConfigMap must then be applied with Salt.

```
root@bootstrap $ kubectl exec --kubeconfig /etc/kubernetes/admin.conf \
                    -n kube-system -c salt-master salt-master-bootstrap -- \
                    salt-run state.sls \
                    metalk8s.addons.prometheus-operator.deployed \
                    saltenv=metalk8s-2.11.7
```

Dex Configuration Customization

Enable or Disable the Static User Store

Dex includes a local store of users and their passwords, which is enabled by default.

Important: To continue using MetalK8s OIDC (especially for MetalK8s UI and Grafana) in case of the loss of external identity providers, it is advised to keep the static user store enabled.

To disable (resp. enable) it, perform the following steps:

1. Set the `enablePasswordDB` configuration flag to `false` (resp. `true`):

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
                    edit configmap metalk8s-dex-config -n metalk8s-auth
```

```
# [...]
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com/v1alpha2
    kind: DexConfiguration
    spec:
      # [...]
      config:
        # [...]
        enablePasswordDB: false  # or true
```

2. Apply your changes:

```
root@bootstrap $ kubectl exec -n kube-system -c salt-master \
                    --kubeconfig /etc/kubernetes/admin.conf \
                    salt-master-bootstrap -- salt-run state.sls \
                    metalk8s.addons.dex.deployed saltenv=metalk8s-2.11.7
```

Note: Dex enables other operations on static users, such as *Adding a Static User*, and *Changing a Static User Password*.

Additional Configurations

All configuration options exposed by Dex can be changed by following a similar procedure to the ones documented above. Refer to Dex documentation for an exhaustive explanation of what is supported.

To define (or override) any configuration option, follow these steps:

1. Add (or change) the corresponding field under the `spec.config` key of the *metalk8s-auth/metalk8s-dex-config* ConfigMap:

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
                    edit configmap metalk8s-dex-config -n metalk8s-auth
```

   For example, registering a client application with Dex can be done by adding a new entry under `staticClients`:

```
# [...]
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com/v1alpha2
    kind: DexConfiguration
    spec:
      # [...]
      config:
        # [...]
        staticClients:
        - id: example-app
          secret: example-app-secret
          name: 'Example App'
          # Where the app will be running.
          redirectURIs:
          - 'http://127.0.0.1:5555/callback'
```

2. Apply your changes by running:

```
root@bootstrap $ kubectl exec -n kube-system -c salt-master \
                    --kubeconfig /etc/kubernetes/admin.conf \
                    salt-master-bootstrap -- salt-run state.sls \
                    metalk8s.addons.dex.deployed saltenv=metalk8s-2.11.7
```

Loki Configuration Customization

Default configuration for Loki can be overridden by editing its Cluster and Service ConfigMap `metalk8s-loki-config` in namespace `metalk8s-logging` under the key `data.config.yaml`:

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
                    edit configmap -n metalk8s-logging \
                    metalk8s-loki-config
```

The following documentation is not exhaustive and is just here to give some hints on basic usage, for more details or advanced configuration, see the official Loki documentation.

Add Loki memory limit

Loki consumes some memory to store chunks before they get written to disks. Its memory consumption really depends on the usage, which is why we do not set any limit by default.

However, if Loki is unable to write to the disk for any reason, it will continue keeping logs in memory, leading to large memory consumption until the issue is resolved. To prevent Loki from taking too much from the host, potentially leading to starvation, you can define a resource limit on the Pod.

For example, to set the limit to 4 GiB, the ConfigMap must be edited as follows:

```
apiVersion: v1
kind: ConfigMap
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com
    kind: LokiConfig
```

(continues on next page)

(continued from previous page)

```
    spec:
      deployment:
        resources:
          limits:
            memory: "4Gi"
```

Changing the logs retention period

Retention period is the time the logs will be stored and available before getting purged.

For example, to set the retention period to 1 week, the ConfigMap must be edited as follows:

```
apiVersion: v1
kind: ConfigMap
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com
    kind: LokiConfig
    spec:
      config:
        table_manager:
          retention_period: 168h
```

Note: Due to internal implementation, `retention_period` must be a multiple of `24h` in order to get the expected behavior

Metalk8s UI Configuration Customization

Default configuration for MetalK8s UI can be overridden by editing its Cluster and Service ConfigMap `metalk8s-ui-config` in namespace `metalk8s-ui` under the key `data.config\.yaml`:

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
                    edit configmap -n metalk8s-ui \
                    metalk8s-ui-config
```

Changing the MetalK8s UI Ingress Path

In order to expose another UI at the root path of the control plane, in place of MetalK8s UI, you need to change the Ingress path from which MetalK8s UI is served.

For example, to serve MetalK8s UI at **/platform** instead of **/**, follow these steps:

1. Change the value of `spec.basePath` in the ConfigMap:

```
apiVersion: v1
kind: ConfigMap
data:
  config.yaml: |-
```

(continues on next page)

(continued from previous page)

```
    apiVersion: addons.metalk8s.scality.com/v1alpha1
    kind: UIConfig
    spec:
      basePath: /platform
```

1. Apply your changes by running:

```
root@bootstrap $ kubectl exec -n kube-system -c salt-master \
                    --kubeconfig /etc/kubernetes/admin.conf \
                    salt-master-bootstrap -- salt-run state.sls \
                    metalk8s.addons.ui.deployed saltenv=metalk8s-2.11.7
```

MetalK8s Shell UI Configuration Customization

Default configuration for MetalK8s Shell UI can be overridden by editing its Cluster and Service ConfigMap `metalk8s-shell-ui-config` in namespace `metalk8s-ui` under the key `data.config\.yaml`.

Changing UI OIDC Configuration

In order to adapt the OIDC configuration (e.g. the provider URL or the client ID) used by the UI shareable navigation bar (called Shell UI), you need to modify its ConfigMap.

For example, in order to replace the default client ID with "ui", follow these steps:

1. Edit the ConfigMap:

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
                 edit configmap -n metalk8s-ui \
                 metalk8s-shell-ui-config
```

1. Add the following entry:

```
apiVersion: v1
kind: ConfigMap
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com/v1alpha1
    kind: ShellUIConfig
    spec:
      # [...]
      oidc:
        # [...]
        clientId: "ui"
```

1. Apply your changes by running:

```
root@bootstrap $ kubectl exec -n kube-system -c salt-master \
                    --kubeconfig /etc/kubernetes/admin.conf \
                    salt-master-bootstrap -- salt-run state.sls \
                    metalk8s.addons.ui.deployed saltenv=metalk8s-2.11.7
```

You can similarly edit the requested scopes through the "scopes" attribute or the OIDC provider URL through the "providerUrl" attribute.

Changing UI Menu Entries

To change the UI navigation menu entries, follow these steps:

1. Edit the ConfigMap:

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
                 edit configmap -n metalk8s-ui \
                 metalk8s-shell-ui-config
```

1. Edit the `options` field. As an example, we add an entry to the `main` section (there is also a `subLogin` section):

```
apiVersion: v1
kind: ConfigMap
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com/v1alpha1
    kind: ShellUIConfig
    spec:
      # [...]
      options:
        # [...]
        main:
          # [...]
          https://www.scality.com/:
            en: "Scality"
            fr: "Scality"
```

1. Apply your changes by running:

```
root@bootstrap $ kubectl exec -n kube-system -c salt-master \
                    --kubeconfig /etc/kubernetes/admin.conf \
                    salt-master-bootstrap -- salt-run state.sls \
                    metalk8s.addons.ui.deployed saltenv=metalk8s-2.11.7
```

Replicas Count Customization

MetalK8s administrators can scale the number of pods for any service mentioned below by changing the number of replicas which is by default set to a single pod per service.

Service	Namespace	ConfigMap
Alertmanager	metalk8s-monitoring	metalk8s-alertmanager-config
Grafana		metalk8s-grafana-config
Prometheus		metalk8s-prometheus-config
Dex	metalk8s-auth	metalk8s-dex-config
Loki	metalk8s-logging	metalk8s-loki-config

To change the number of replicas, perform the following operations:

1. From the Bootstrap node, edit the `ConfigMap` attributed to the service and then modify the replicas entry.

```
root@bootstrap $ kubectl --kubeconfig /etc/kubernetes/admin.conf \
                    edit configmap <ConfigMap> -n <Namespace>
```

Note: For each service, consult the *Cluster Services* table to obtain the `ConfigMap` and the `Namespace` to be used for the above command.

Make sure to replace **<number-of-replicas>** field with an integer value (For example 2).

```
[...]
data:
   config.yaml: |-
      spec:
         deployment:
            replicas: <number-of-replicas>
[...]
```

2. Save the ConfigMap changes.
3. From the Bootstrap node, execute the following command which connects to the Salt master container and applies salt-states to propagate the new changes down to the underlying services.

```
root@bootstrap $ kubectl exec --kubeconfig /etc/kubernetes/admin.conf \
                  -n kube-system -c salt-master salt-master-bootstrap \
                  -- salt-run state.sls metalk8s.deployed \
                  saltenv=metalk8s-2.11.7
```

Note: Scaling the number of pods for services like `Prometheus`, `Alertmanager` and `Loki` requires provisioning extra persistent volumes for these pods to startup normally. Refer to *this procedure* for more information.

2.4 Volume Management

This section covers MetalK8s volume management operations, from creating a StorageClass, to creating and deleting a volume using the CLI or the UI. Volumes enable the use of persistent data storage within a MetalK8s Cluster.

2.4.1 StorageClass Creation

MetalK8s uses StorageClass objects to describe how volumes are formatted and mounted. This topic explains how to use the CLI to create a StorageClass.

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:
```

(continues on next page)

(continued from previous page)

```
  fsType: <filesystem_type>
  mkfsOptions: <mkfs_options>
```

Set the following fields:

- `mountOptions`: specifies how the volume should be mounted. For example: `rw` (read/write), or `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
```

2.4.2 Volume Management Using the CLI

This topic describes how to create and delete a MetalK8s Volume using the CLI. Volume objects enable a declarative provisioning of persistent storage, to use in Kubernetes workloads (through PersistentVolumes).

Requirements

- StorageClass objects must be registered in your cluster to create Volumes. For more information refer to *StorageClass Creation*.

Creating a Volume

1. Create a Volume manifest using one of the following templates:

rawBlockDevice Volumes

```
apiVersion: storage.metalk8s.scality.com/v1alpha1
kind: Volume
metadata:
  name: <volume_name>
spec:
  nodeName: <node_name>
  storageClassName: <storageclass_name>
```

(continues on next page)

(continued from previous page)

```
  mode: "Filesystem"
  rawBlockDevice:
    devicePath: <devicePath>
```

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.
- **mode**: describes how the volume is intented to be consumed, either Block or Filesystem (default to Filesystem if not specified).
- **devicePath**: path to the block device (for example: `/dev/sda1`).

lvmLogicalVolume Volumes

```
apiVersion: storage.metalk8s.scality.com/v1alpha1
kind: Volume
metadata:
  name: <volume_name>
spec:
  nodeName: <node_name>
  storageClassName: <storageclass_name>
  mode: "Filesystem"
  lvmLogicalVolume:
    vgName: <vg_name>
    size: 10Gi
```

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.
- **mode**: describes how the volume is intented to be consumed, either Block or Filesystem (default to Filesystem if not specified).
- **vgName**: LVM VolumeGroup name to create the LogicalVolume the VolumeGroup must exists on the Node.
- **size**: Size of the LVM LogicalVolume to create.

2. Create the Volume.

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

3. Check that the Volume has been created.

```
root@bootstrap $ kubectl get volume <volume_name>
NAME              NODE        STORAGECLASS
<volume_name>   bootstrap   metalk8s-demo-storageclass
```

Deleting a Volume

Note: A Volume object can only be deleted if there is no backing storage, or if the volume is not in use. Otherwise, the volume will be marked for deletion and remain available until one of the conditions is met.

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
```

2.4.3 Volume Management Using the UI

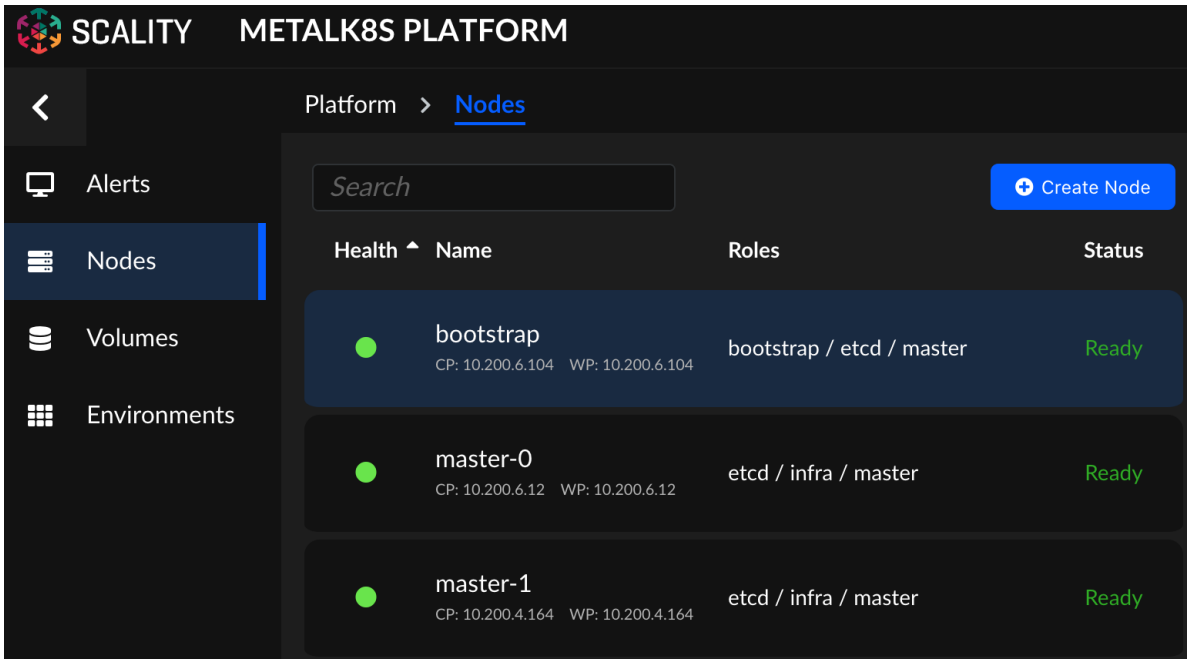
This topic describes how to create and delete a MetalK8s Volume using the MetalK8s UI.

Requirements

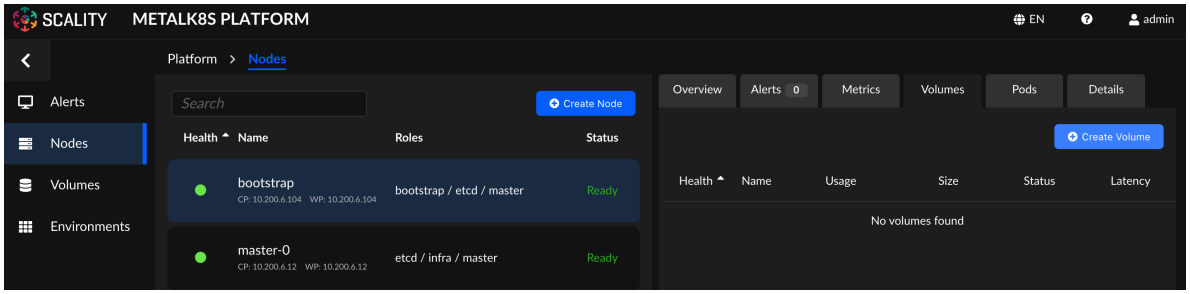
- StorageClass objects must be registered in your cluster to create Volumes. For more information refer to *StorageClass Creation*.
- Access the MetalK8s UI. Refer to *this procedure*.

Creating a Volume

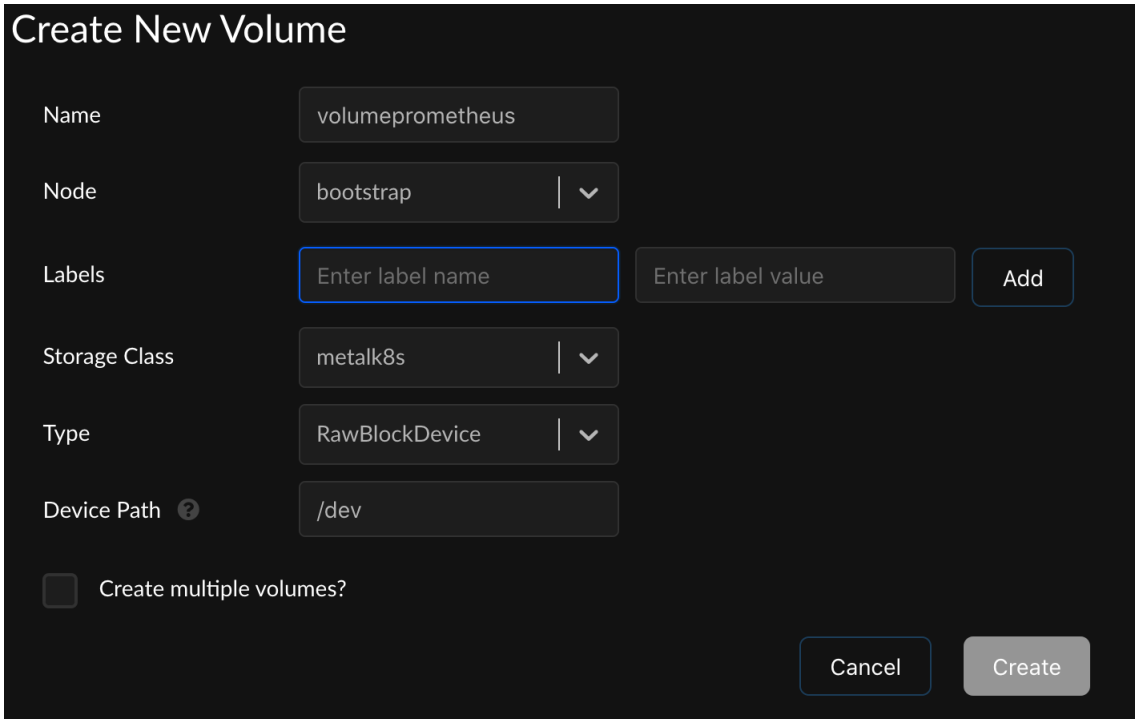
1. Click **Nodes** on the sidebar to access the node list.



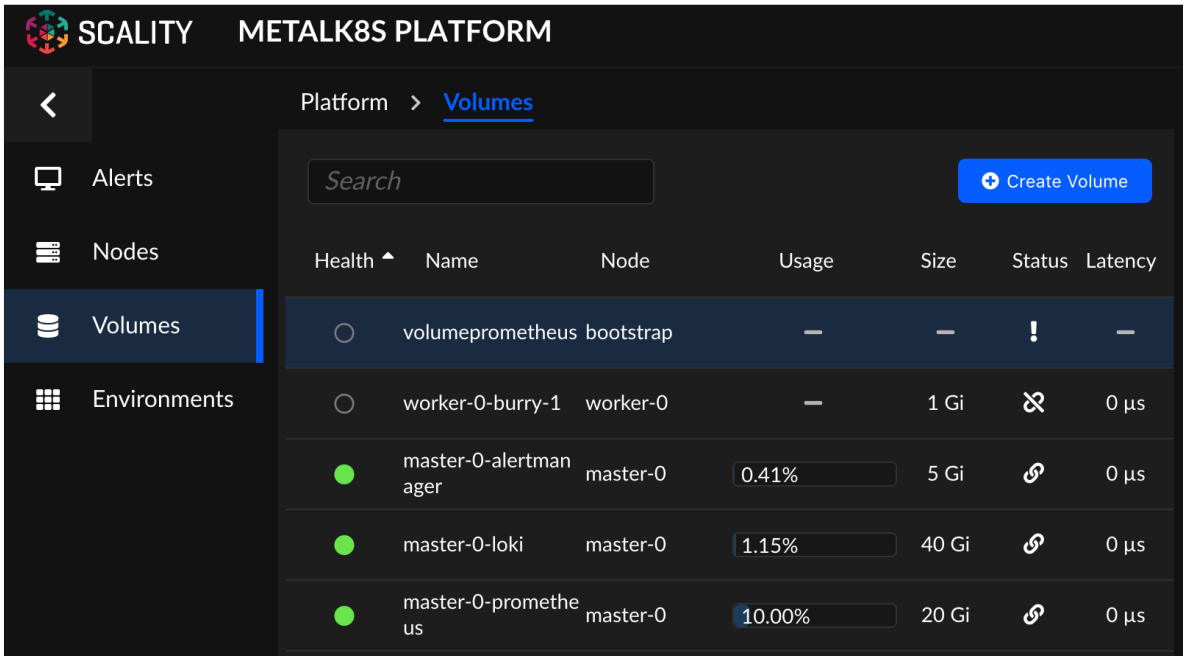
2. On the node list, select the node you want to create a volume on.
3. Go to the **Volumes** tab and click **+ Create Volume**.



4. Fill in the respective fields, and click **Create**.
   - **Name**: Denotes the volume name.
   - **Labels**: A set of key/value pairs used by PersistentVolumeClaims to select the right PersistentVolumes.
   - **Storage Class**: Refers to *StorageClass Creation*.
   - **Type**: MetalK8s currently only supports RawBlockDevice and SparseLoopDevice.
   - **Device path**: Refers to the path of an existing storage device.

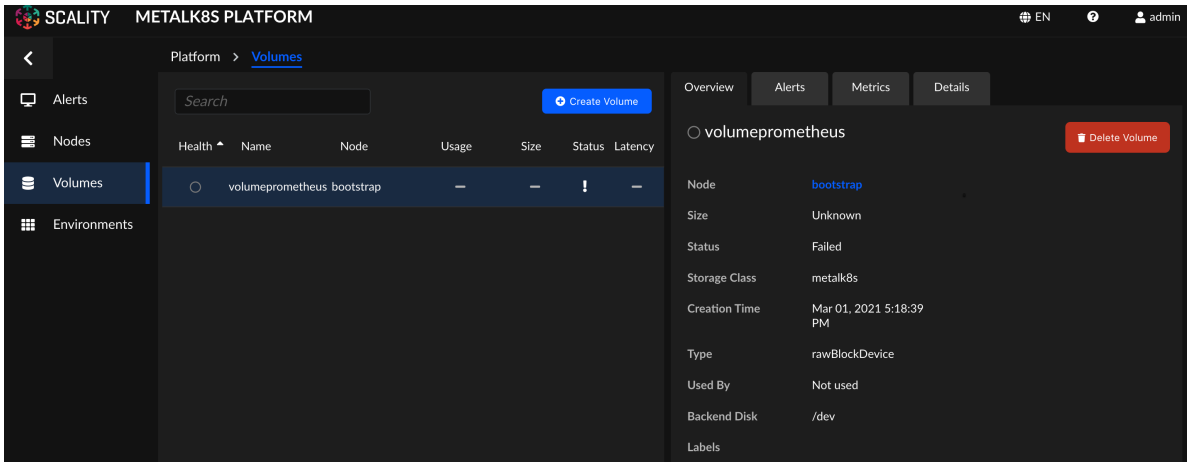


5. Click **Volumes** on the sidebar to access the volume list. The new volume created appears in the list.

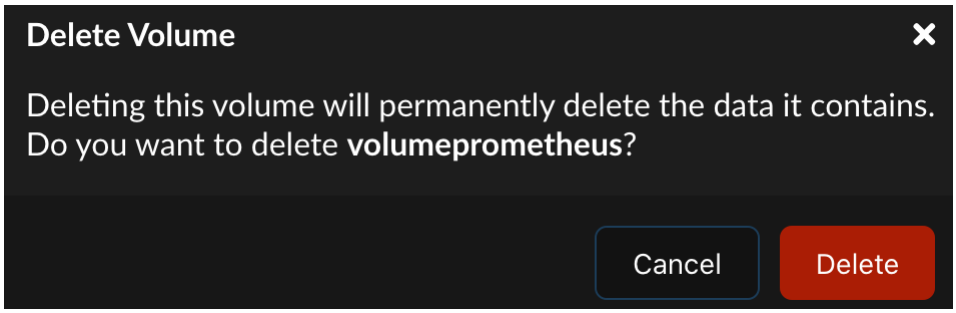


Deleting a Volume

1. Click **Volumes** on the sidebar to access the volume list, and select the volume you want to delete.
2. Go to the **Overview** tab, click **Delete Volume**.



3. Confirm the volume deletion request by clicking **Delete**.



2.5 Cluster Upgrade

MetalK8s clusters are upgraded using the utility scripts packaged with every new release. This topic describes upgrading MetalK8s and all components included in the stack.

2.5.1 Supported Versions

Note: MetalK8s supports upgrade of *at most* one minor version at a time. For example:

- from 2.4.0 to 2.4.4,
- from 2.4.0 to 2.5.1.

Refer to the release notes for more information.

2.5.2 Prerequisites

ISO Preparation

Provision a new MetalK8s ISO by running the utility script shipped with the current installation.

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

Pre-Checks

Use the `--dry-run` option to validate your environment for upgrade:

```
/srv/scality/metalk8s-X.Y.Z/upgrade.sh --dry-run --verbose
```

This will simulate the upgrade pre-checks and provide an overview of the changes to be carried out in your MetalK8s cluster.

Important: The version prefix metalk8s-**X.Y.Z** must be the *new* MetalK8s version you want to upgrade to.

2.5.3 Upgrade

1. Run the utility script shipped with the *new* version you want to upgrade to.
2. From the *Bootstrap node*, launch the upgrade.

   ```
   /srv/scality/metalk8s-X.Y.Z/upgrade.sh
   ```

   Important: The version prefix metalk8s-**X.Y.Z** must be the *new* MetalK8s version you want to upgrade to.

2.6 Cluster Downgrade

MetalK8s clusters are downgraded using the utility scripts that are packaged with your current installation. This topic describes downgrading MetalK8s and all components included in the stack.

2.6.1 Supported Versions

Note: MetalK8s supports downgrade of **at most** one minor version at a time. For example:

- from 2.4.4 to 2.4.1,
- from 2.5.1 to 2.4.0.

Refer to the release notes for more information.

Warning: Version only supports downgrade of patch version.

2.6.2 Prerequisites

ISO Preparation

Provision a new **MetalK8s** ISO by running the utility script shipped with the current installation.

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

Pre-Checks

You can test if your environment will successfully downgrade with the following command.

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

This will simulate the downgrade pre-checks and provide an overview of the changes to be carried out in your MetalK8s cluster.

Important: The version prefix metalk8s-**X.Y.Z** must be the *current* installed MetalK8s version.

2.6.3 Downgrade

1. Run the utility script shipped with the *current* installation providing it with the destination version.
2. From the *Bootstrap node*, launch the downgrade.

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

Important: The version prefix metalk8s-**X.Y.Z** must be the *current* installed MetalK8s version.

2.7 Disaster Recovery

This section offers a series of recovery operations such as the backup and restoration of the MetalK8s bootstrap node.

2.7.1 Bootstrap Node Backup and Restoration

This topic describes how to back up a MetalK8s bootstrap node manually, and how to restore a bootstrap node from such a backup.

Note: A backup is generated automatically:

- at the end of the bootstrap,
- at the beginning of an upgrade or downgrade,
- at the end of an upgrade or downgrade,
- at the end of a bootstrap restoration.

Backing Up a Bootstrap Node

To create a new backup file, run the following command:

`/srv/scality/metalk8s-2.11.7/backup.sh`

Backup archives are stored in /var/lib/metalk8s/backups on all master nodes.

Restoring a Bootstrap Node

Warning: You must have a highly available control plane with at least three members in the etcd cluster (including the failed bootstrap node), to use the restore script.

Note: To restore a bootstrap node you need a backup archive and MetalK8s ISOs. All the ISOs referenced in the bootstrap configuration file (located in /etc/metalk8s/bootstrap.yaml) must be present.

---

1. Unregister the unreachable etcd member from the cluster by running the following commands from a working node with the etcd role:
   1. Get etcd container id.

      ```
      CONT_ID=$(crictl ps -q --label io.kubernetes.container.name=etcd --state␣
      ↪Running)
      ```

   2. List all etcd members to get the ID of the etcd member that needs to be removed.

      ```
      crictl exec -it "$CONT_ID" \
         etcdctl --endpoints https://localhost:2379 \
         --cacert /etc/kubernetes/pki/etcd/ca.crt \
         --key /etc/kubernetes/pki/etcd/server.key \
         --cert /etc/kubernetes/pki/etcd/server.crt \
         member list
      ```

   3. Remove the etcd member (replace <etcd_id> in the command).

```
crictl exec -it "$CONT_ID" \
   etcdctl --endpoints https://localhost:2379 \
   --cacert /etc/kubernetes/pki/etcd/ca.crt \
   --key /etc/kubernetes/pki/etcd/server.key \
   --cert /etc/kubernetes/pki/etcd/server.crt \
   member remove <etcd_id>
```

2. Because multiple bootstrap nodes are not supported, remove the old bootstrap node before performing the restoration by running the following commands from a working node with a master role:

   1. List all nodes to get the node name of the old bootstrap node that needs to be removed.

   ```
   kubectl get node --selector="node-role.kubernetes.io/bootstrap" \
      --kubeconfig=/etc/kubernetes/admin.conf
   ```

   2. Remove the old bootstrap node (replace <node_name> in the command).

   ```
   kubectl delete node <node_name> --kubeconfig=/etc/kubernetes/admin.conf
   ```

3. Mount the ISO.

4. Restore the bootstrap node. Replace <backup_archive> with the path to the backup archive you want to use, and <node_ip> with a control plane IP of one control plane node.

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

2.8 Solution Deployment

To deploy a solution in a MetalK8s cluster, a utility script is provided. This procedure describes how to deploy a solution using this tool, which is located at the root of the MetalK8s archive:

```
/srv/scality/metalk8s-2.11.7/solutions.sh
```

2.8.1 Preparation

1. Import a solution in the cluster, and make the container images available through the cluster registry.

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

2. Activate a solution version.

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

   Only one version of a solution can be active at a time. An active solution version provides cluster-wide resources, such as CRDs, to all other versions of this solution.

2.8.2 Deployment

1. Solutions are meant to be deployed in isolated namespaces called *environments*.

   To create an environment, run:

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

2. Solutions are packaged with an *Operator* to provide all required domain-specific logic. To deploy a solution operator in an environment, run:

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

2.8.3 Configuration

The solution operator is now deployed. To finalize the deployment and configuration of a solution, refer to its documentation.

2.9 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: '2.11.7'
    <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.
↪11.7   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
```

2.10 Changing the Control Plane Ingress IP

This procedure describes how to change the Control Plane Ingress IP, and to enable (or disable) MetalLB management of this IP.

Note: Disabling MetalLB using this procedure does **not** remove MetalLB, it simply disables its use for managing the `LoadBalancer` *Service* for MetalK8s Control Plane Ingress.

---

1. On the Bootstrap node, update the `ip` field from `networks.controlPlane.ingress` in the Bootstrap configuration file. (refer to *Bootstrap Configuration*)
2. Refresh the pillar.

   ```
   $ salt-call saltutil.refresh_pillar wait=True
   ```

3. Check that the change is taken into account.

   ```
   $ salt-call metalk8s_network.get_control_plane_ingress_ip
   local:
       <my-new-ip>
   $ salt-call pillar.get networks:control_plane
   local:
     ----------
     cidr:
       - <control-plane-cidr>
     ingress:
       ip:
         <my-new-ip>
     metalLB:
       enabled: <true | false>
   ```

4. On the Bootstrap node, reconfigure apiServer:

   ```
   $ salt-call state.sls \
       metalk8s.kubernetes.apiserver \
       saltenv=metalk8s-2.11.7
   ```

5. Reconfigure Control Plane components:

   ```
   $ kubectl exec -n kube-system -c salt-master \
       --kubeconfig=/etc/kubernetes/admin.conf \
       $(kubectl --kubeconfig=/etc/kubernetes/admin.conf get pod \
       -l "app.kubernetes.io/name=salt-master" \
       --namespace=kube-system -o jsonpath='{.items[0].metadata.name}')  \
       -- salt-run state.orchestrate \
       metalk8s.orchestrate.update-control-plane-ingress-ip \
       saltenv=metalk8s-2.11.7
   ```

6. You can *access the MetalK8s GUI* using this new IP.

2.11 Using the `metalk8s-utils` Image

A MetalK8s installation comes with a container image called `metalk8s-utils` in the embedded registry. This image contains several tools an operator can use to analyze a cluster environment or troubleshoot various system issues.

The image can be used to create a *Pod* on a node, after which a shell inside the container can be created to run the various utilities. Depending on the use-case, the *Pod* could be created using the host network namespace, the host PID namespace, elevated privileges, mounting host directories as volumes, etc.

See the `metalk8s-utils` `Dockerfile` for a list of all packages installed in the image.

2.11.1 A Simple Shell

To run a `metalk8s-utils` container as a simple shell, execute the following command:

```
kubectl run shell \
    --image=metalk8s-registry-from-config.invalid/metalk8s-2.11.7/metalk8s-utils:2.11.7 \
    --restart=Never \
    --attach \
    --stdin \
    --tty \
    --rm
```

This will create a *Pod* called `shell` with a container running the `metalk8s-utils` image, and present you with a shell in this container.

Note: This procedure expects no other `shell` *Pod* to be running. Adjust the name accordingly, or use a dedicated namespace if conflicts occur.

2.11.2 A Long-Running Container

In the example above, the lifetime of the container is tied to the invocation of `kubectl run`. In some situations it's more efficient to keep such container running and attach to it (and detach from it) dynamically.

- Create the *Pod*:

```
kubectl run shell \
    --image=metalk8s-registry-from-config.invalid/metalk8s-2.11.7/metalk8s-utils:2.11.7 \
    --restart=Never \
    --command -- sleep infinity
```

This creates the `shell` *Pod* including a `metalk8s-utils` container running `sleep infinity`, effectivelly causing the *Pod* to remain alive until deleted.

- Get a shell in the container:

```
kubectl exec -ti shell -- bash
```

Note: The **`screen`** and **`tmux`** utilities are installed in the image for terminal multiplexing.

- Exit the shell to detach
- Remove the *Pod* once the container is no longer needed:

```
kubectl delete pod shell
```

2.11.3 A Shell on a Particular Node

To pin the *Pod* in which the `metalk8s-utils` container is launched to a particular node, add the following options to a suitable `kubectl run` invocation:

```
--overrides='{ "apiVersion": "v1", "spec": { "nodeName": "NODE_NAME" } }'
```

Note: In the above, replace `NODE_NAME` by the desired node name.

2.11.4 A Shell in the Host Network Namespace

To run a `metalk8s-utils` container in the host network namespace, e.g., to use utilities such as `ip`, `iperf` or `tcpdump` as if they're executed on the host, add the following options to a suitable `kubectl run` invocation:

```
--overrides='{ "apiVersion": "v1", "spec": { "hostNetwork": true } }'
```

Note: If multiple `overrides` need to be combined, the JSON objects must be merged.

2.12 Registry HA

To be able to run fully offline, MetalK8s comes with its own registry serving all necessary images used by its containers. This registry container sits on the Bootstrap node.

With a highly available registry, container images are served by multiple nodes, which means the Bootstrap node can be lost without impacting the cluster. It allows pods to be scheduled, even if the needed images are not cached locally.

Note: This procedure only talk about registry HA as Bootstrap HA is not supported for the moment, so it's only a part of the Bootstrap functionnaly. Check this ticket for more informations https://github.com/scality/metalk8s/issues/2002

2.12.1 Prepare the node

To configure a node to host a registry, a `repository` pod must be scheduled on it. This node must be part of the MetalK8s cluster and no specific roles or taints are needed.

All ISOs listed in the `archives` section of `/etc/metalk8s/bootstrap.yaml` and `/etc/metalk8s/solutions.yaml` must be copied from the Bootstrap node to the target node at exactly the same location.

2.12.2 Deploy the registry

Connect to the node where you want to deploy the registry and run the following salt states

- Prepare all the MetalK8s ISOs

```
root@node-1 $ salt-call state.sls \
    metalk8s.archives.mounted \
    saltenv=metalk8s-2.11.7
```

- If you have some solutions, prepare the solutions ISOs

```
root@node-1 $ salt-call state.sls \
    metalk8s.solutions.available \
    saltenv=metalk8s-2.11.7
```

- Deploy the registry container

```
root@node-1 $ salt-call state.sls \
    metalk8s.repo.installed \
    saltenv=metalk8s-2.11.7
```

2.12.3 Reconfigure the container engines

Containerd must be reconfigured to add the freshly deployed registry to its endpoints and so it can still pull images in case the Bootstrap node's one is down.

From the Bootstrap node, run (replace <bootstrap_node_name> with the actual Bootstrap node name):

```
root@bootstrap $ kubectl exec -n kube-system -c salt-master \
   --kubeconfig=/etc/kubernetes/admin.conf \
   salt-master-<bootstrap_node_name> -- salt '*' state.sls \
   metalk8s.container-engine saltenv=metalk8s-2.11.7
```

2.13 Listening Processes

In MetalK8s context several processes are deployed and they need to communicate with each other, sometimes locally, sometimes between machines in the cluster, or with the end user.

Depending on their *roles*, nodes must have several addresses available for MetalK8s processes to bind.

2.13.1 Listening Processes on Bootstrap Nodes

Address	Description
control_plane_ip:4505	Salt master publisher
control_plane_ip:4506	Salt master request server
control_plane_ip:4507	Salt API
control_plane_ip:8080	MetalK8s repository

2.13.2 Listening Processes on Master Nodes

Address	Description
control_plane_ip:6443	Kubernetes apiserver
127.0.0.1:7080	Apiserver proxy health check
127.0.0.1:7443	Apiserver proxy
control_plane_ip:7472	Control plane MetalLB speaker metrics (only if MetalLB enabled)
control_plane_ip:7946	Control plane MetalLB speaker (only if MetalLB enabled)
ingress_control_plane_ip:8443	Control plane nginx ingress
control_plane_ip:10257	Kubernetes controller manager
control_plane_ip:10259	Kubernetes scheduler

2.13.3 Listening Processes on Etcd Nodes

Address	Description
127.0.0.1:2379	Etcd client
control_plane_ip:2379	Etcd client
control_plane_ip:2380	Etcd peer
127.0.0.1:2381	Etcd metrics
control_plane_ip:2381	Etcd metrics

2.13.4 Listening Processes on All Nodes

Address	Description
127.0.0.1:9099	Calico node
control_plane_ip:9100	Node exporter
127.0.0.1:10248	Kubelet health check
control_plane_ip:10249	Kubernetes proxy metrics
control_plane_ip:10250	Kubelet
control_plane_ip:10256	Kubernetes proxy health check

2.14 Troubleshooting

This section covers some of the common issues users face during and after a MetalK8s operation.

If your issue is not presented here, create a GitHub issue or open a new GitHub discussion.

2.14.1 Account Administration Errors

Forgot the MetalK8s GUI Password

If you forget the MetalK8s GUI user name or password, refer to *Changing Static User Password* to reset or change your credentials.

2.14.2 General Kubernetes Resource Errors

Pod Status Shows CrashLoopBackOff

If some pods are in a persistent **CrashLoopBackOf** state, it means that the pods are crashing because they start up then immediately exit. Kubernetes restarts them and the cycle continues. To find potential causes of this error, review the output returned from the following command:

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

Persistent Volume Claim (PVC) Stuck in Pending State

If after provisioning a volume for a pod (for example Prometheus) the PVC still hangs in a **Pending** state, perform the following checks:

1. Check that the volumes have been provisioned and are in a **Ready** state.

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

2. Check that a corresponding PersistentVolume exists.

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

3. Check that the PersistentVolume matches the PersistentVolumeClaim constraints (size, labels, storage class).

   - Find the name of your PersistentVolumeClaim:

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

- Check if the PersistentVolumeClaim constraints match:

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

4. If no PersistentVolume exists, check that the storage operator is up and running.

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

Access to MetalK8s GUI Fails With "undefined backend"

If you encounter an "undefined backend" error while using the MetalK8s GUI, perform the following checks:

1. Check that the ingress controller pods are running.

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

2. Check the ingress controller logs.

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

2.15 Sosreport

The sosreport tool is installed automatically on all MetalK8s hosts, and embeds some custom plugins. It allows to generate a report from a host, including logs, configurations, containers information, etc. This report can then be consumed by an operator, or shared with Scality support, to investigate problems on a platform.

2.15.1 Generate A Report

To generate a report for a machine, you must have root access.

To include logs and configuration for containerd and MetalK8s components, run:

```
root@your-machine # sosreport --batch --all-logs \
    -o metalk8s -kmetalk8s.all=True -kmetalk8s.podlogs=True -kmetalk8s.describe=True \
    -o containerd -kcontainerd.all=True -kcontainerd.logs=True
```

The name of the generated archive is printed in the console output.

2.15.2 Plugins List

To display the full list of available plugins and their options, run:

```
sosreport --list-plugins
```

CHAPTER

THREE

DEVELOPER GUIDE

3.1 Architecture Documents

3.1.1 Alert History

Context

In NextGen UI we are introducing the Global Health Component that shows real time entity Health but also intends to show entity Health over the last X days. This Global Health Component is available in the System Health Monitor as well as in Metalk8s, xCore and XDM admin UIs. It applies to entities like Node, Volume, Platform, Storage Backends, etc.



The entity Health is computed based on active alerts. In order to know the health of an entity as it was in the past, we would need to collect alerts that were active for this specific past time. As soon as the Platform or Storage Admin identifies a time at which the entity was degraded, he can access the detailed list of sub alerts impacting this entity.

Currently, once an active alert is cleared, it disappears from the System.

Goal

In order to achieve the UI functionality as described above, we would need to keep information about the alerts that were fired in the past:

- The alert (all info that were available at the time the alert was fired)
- When it was fired
- When it was cleared

User Stories

As a Platform/Storage Admin, I want to know the health of a given NextGen entity over the past X days in order to ease root cause analysis.

Basically this should be achieved by collecting past alerts, belonging to this entity.

As a Platform/Storage Admin, I want to collect the list of sub alerts which contributed to the degradation on a specific entity in the past, in order to understand more in details the cause of the degradation.

Here we would need to access all sub alerts (contributing to the entity high level alert). This is related to the Alert grouping feature.

The X days to keep accessible is configurable and ideally matches with the history of other observability data (metrics and logs) in order to ease the correlation between various observability indicators. This configuration must be persistent across platform upgrades.

In conclusion, the system should retain all emitted alerts for a given configurable period.

The service exposing past alerts is to be used by NextGen Admin UIs. It can also be used by some NextGen tooling when it comes to create a support ticket. It will not be used by xCore or XDM data workloads and it will not be exposed for external usage.

Monitoring and Alerting

The service exposing past alerts should be monitored i.e. should expose key health/performance indicators that one can consume through dedicated Grafana dashboard. An alert should be triggered when the service is degraded.

Deployment

The said service is part of the infra service category and it is either deployed automatically or some documentation explains how to deploy it and provision storage for it.

It should support one node failure when deploying NextGen on more than 3 nodes, like for monitoring, alerting and logging services.

Future/Bonus Features

Dedicated Grafana Dashboard to navigate through the past alerts without focusing on a specific entity only. From this dashboard, one can select one or multiple labels as well as a specific period, in order to collect all alerts with a given set of labels.

A dump of the past alerts could be added to the sos report that one would generate when collecting all information to send to Scality support.

Design Choices

Alertmanager webhook

To retrieve alerts sent by Alertmanager, we configure a specific receiver where it sends each and every incoming alerts. This receiver is a webhook which is basically an HTTP server listening on a port and waiting for HTTP POST request from Alertmanager. It then forwards alerts to the storage backend.

Alerts sent by Alertmanager are JSON formatted as follows:

```
{
  "version": "4",
  "groupKey": <string>,              // key identifying the group of alerts (e.g. to␣
↪deduplicate)
  "truncatedAlerts": <int>,          // how many alerts have been truncated due to "max_
↪alerts"
  "status": "<resolved|firing>",
  "receiver": <string>,
  "groupLabels": <object>,
  "commonLabels": <object>,
  "commonAnnotations": <object>,
  "externalURL": <string>,           // backlink to the Alertmanager.
  "alerts": [
    {
      "status": "<resolved|firing>",
      "labels": <object>,
      "annotations": <object>,
      "startsAt": "<rfc3339>",
      "endsAt": "<rfc3339>",
      "generatorURL": <string>       // identifies the entity that caused the alert
    },
    ...
  ]
}
```

Alertmanager implements an exponential backoff retry mechanism, so We can not miss alerts if the webhook is unreachable/down. It will keep retrying until it manages to send the alerts.

Loki as storage backend

We use Loki as the storage backend for alert history because it provides several advantages.

First, it allows to easily store the alerts by simply logging them on the webhook container output, letting Fluent-bit forward the alerts to it.

Loki uses a NoSQL database, which is better to store JSON documents than an SQL one, allowing us to not have to create and maintain a database schema for the alerts.

Loki also provides an API allowing us to expose and query these alerts using the LogQL language.

Plus, since Loki is already part of the cluster, it saves us from having to install, manage and expose a new database.

Using Loki, we also directly benefit from its retention and purge mechanisms, making the alerts history retention time automatically aligned with all other logs (14 days by default).

> **Warning:** There is a drawback in using Loki, if at some point its volume is full (because there is too much logs), we will not be able to store new alerts anymore, especially since there is no size-based purge mechanism.
>
> Another issue is, since we share the retention configuration with the other logs, it is hard to ensure we will keep enough alert history.
>
> As for now, there is no retention based on labels, streams, tenant or whatever (on-going discussion GH Loki #162).

Rejected Design Choices

Alertmanager API scraper

A program polling the Alertmanager API to retrieve alerts.

It generates more load and forces us to parse the result from the API to keep track of what we already forward to the storage backend or query it to retrieve the previously sent alerts.

Plus, it does not allow to have alerts in near to real time, except if we poll the API in a really aggressive manner.

If the scraper is down for a long period of time, we could also loose some alerts.

Dedicated database as storage backend

Using a dedicated database to store alerts history was rejected, because it means adding an extra component to the stack.

Furthermore, we would need to handle the database replication, lifecycle, etc.

We would also need to expose this database to the various components consuming the data, probably through an API, bringing another extra component to develop and maintain.

Implementation Details

Alertmanager webhook

We need a simple container, with a basic HTTP server running inside, simply handling POST requests and logging them on the standard output.

It will be deployed by Salt as part of the monitoring stack.

A deployment with only 1 replica will be used as we do not want duplicated entries and Alertmanager handles retry mechanism if the webhook is unreachable.

An example of what we need can be found *here <https://github.com/tomtom-international/alertmanager-webhook-logger>*.

Alertmanager configuration

The default Alertmanager's configuration must be updated to send all alerts to this webhook.

Configuration example:

```
receivers:
  - name: metalk8s-alert-logger
    webhook_configs:
      - send_resolved: true
        url: http://<webhook-ip>:<webhook-port>
route:
  receiver: metalk8s-alert-logger
  routes:
    - receiver: metalk8s-alert-logger
      continue: True
```

This configuration must not be overwritable by any user customization and the `metalk8s-alert-logger` receiver must be the first route to ensure it will receive all the alerts.

Fluent-bit configuration

Logs from the webhook need to be handled differently than the other Kubernetes containers. Timestamps of the logs must be extracted from the JSON `timestamp` key and only the JSON part of the log must be stored to make it easier to use by automatic tools.

Expose Loki API

The Loki API must be reachable via the web UI, therefore it must be exposed through an ingress as it is already done for Prometheus or Alertmanager APIs.

Global Health Component Implementation

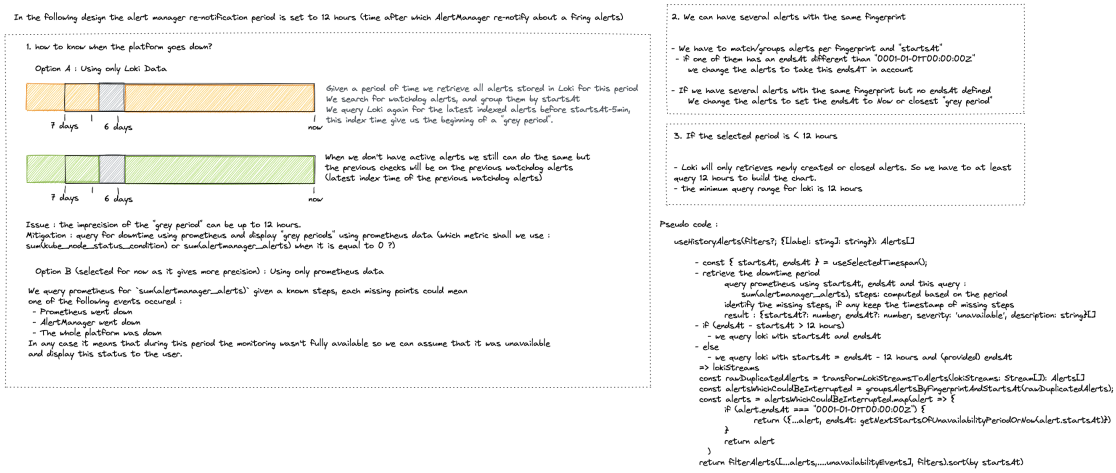
In order to build the Global Health Component the UI queries loki API to retrieve past alerts. The users have the ability to select a timespan for which they want to retrieve the alerts. The UI is using this timespan to query loki for alerts firing during this period. However Alertmanager is repeating webhooks for long running alerts to metalk8s-alert-logger at a defined pace in its configuration. This means that for example if an alert is firing since 6hours and alertmanager is configured to repeat the notification each 12 hours, querying loki for the last hour won't list that alert. Additionnaly if alertmanager or loki or the platform itself goes down old alerts won't be closed and new ones will be fired by alertmanager.

This leads to several issues that the UI have to solve when querying Loki to display the alert history: #. When the platform/alertmanager/loki goes down the UI has to compute the end time of an alert and set it to the end time period start timestamp. #. The alerts are duplicated in loki so the UI has to regroup them by fingerprint and start time.

We propose here to implement a useHistoryAlert hook in metalk8s ui which can be reused to retrieve past alerts. This useHistoryAlert takes a list of filters as a parameter. These filters allow consumers to search for alert history of alerts matching a specific set of labels or annotations. The signature of the hook is: *useHistoryAlert(filters: {[label: string]: string}): Alert[]*.

This hook is used in conjonction with an *AlertHistoryProvider* which is responsible of alert fetching and transformation. It uses *useMetricsTimeSpan* hook to retrieve the period selected by the user and uses this period to fetch alerts on loki. It additionnaly retriveves platform/monitoring unavailbility periods by querying for alertmanager number of firing alerts metrics on Prometheus API. If the selected timespan is smaller than alertmanager notification period (period after which alertmanager recall Metalk8s alert logger to signify that the alert is still firing), the hook is then fetching the alerts for this period of time at minimum to ensure long firing alerts retrieval.

Once the downtime periods are retrieved they are converted to an UI alert object with a specific severity set to *unavailable* so that we can display an unavailable segment on the Global health bar.



Grafana dashboard

Alerts are already retrievable from the `Logs` dashboard, but it is not user friendly as the webhook pod name must be known by the user and metrics displayed are relative to the pod, not the alerts themselves.

A dedicated Grafana dashboard with the alerts and metrics related to them will be added.

This dashboard will be deployed by adding a ConfigMap `alert-history-dashboard` in Namespace `metalk8s-monitoring`:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: alert-history-dashboard
  namespace: metalk8s-monitoring
  labels:
    grafana_dashboard: "1"
data:
  alert-history.json: <DASHBOARD DEFINITION>
```

Test Plan

Add a test in post-install to ensure we can at least retrieve the `Watchdog` alert using Loki API.

3.1.2 Alerting Functionalities

Context

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

Requirements

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

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

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

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

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

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

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

Note: Alertmanager configuration documentation is available here

Design Choices

To be able to edit existing rules, add new ones, etc., and in order to keep these changes across restorations, upgrades and downgrades, we need to put in place some mechanisms to configure Prometheus and Alertmanager and persist these configurations.

For the persistence part, we will rely on what has been done for *CSC* (Cluster and Services Configurations), and use the already defined resources for *Alertmanager* and *Prometheus*.

Extra Prometheus Rules

We will use the already existing `metalk8s-prometheus-config` *ConfigMap* to store the *Prometheus* configuration customizations.

Adding extra alert and record rules will be done editing this *ConfigMap* under the `spec.extraRules` key in `config.yaml` as follows:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: metalk8s-prometheus-config
  namespace: metalk8s-monitoring
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com
    kind: PrometheusConfig
    spec:
      deployment:
```

(continues on next page)

(continued from previous page)

```
    replicas: 1
  extraRules:
    groups:
    - name: <rulesGroupName>
      rules:
      - alert: <AlertName>
        annotations:
          description: description of what this alert is
        expr: vector(1)
        for: 10m
        labels:
          severity: critical
      - alert: <AnotherAlertName>
        [...]
      - record: <recordName>
        [...]
    - name: <anotherRulesGroupName>
      [...]
```

PromQL is to be used to define `expr` field.

This `spec.extraRules` entry will be used to generate through Salt a `PrometheusRule` object named `metalk8s-prometheus-extra-rules` in the `metalk8s-monitoring` namespace, which will be automatically consumed by the *Prometheus* Operator to generate the new rules.

A CLI and UI tooling will be provided to show and edit this configuration.

Edit Existing Prometheus Alert Rules

To edit existing *Prometheus* rules, we can't only define new `PrometheusRules` resources since *Prometheus* Operator will not overwrite those already existing, but will rather append them to the list of rules, ending up with 2 rules with the same name but different parameters.

We also can't edit the `PrometheusRules` deployed by MetalK8s, otherwise we would lose these changes in case of cluster restoration, upgrade or downgrade.

So, in order to allow the user to customize the alert rules, we will pick up some of them (the most relevant ones) and expose only few parts of their configurations (e.g. threshold) to be customized.

It also makes the customization of these alert rules easier for the user as, for example, he will not need to understand PromQL to adapt the threshold of an alert rule.

Since in *Prometheus* rules, there are duplicated group name + alert rule name, we also need to take the severity into account to understand which specific alert we're editing.

These customization will be stored in the `metalk8s-prometheus-config` *ConfigMap* with something like:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: metalk8s-prometheus-config
  namespace: metalk8s-monitoring
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com
```

(continues on next page)

(continued from previous page)

```
  kind: PrometheusConfig
  spec:
    deployment:
      replicas: 1
    rules:
      <alertGroupName>:
        <alertName>:
          warning:
            threshold: 30
          critical:
            threshold: 10
      <anotherAlertGroupName>:
        <anotherAlertName>:
          critical:
            threshold: 20
            anotherThreshold: 10
```

The `PrometheusRules` object manifests `salt/metalk8s/addons/prometheus-operator/deployed/chart.sls` need to be templatized to consume these customizations through CSC module.

Default values for customizable alert rules to fallback on, if not defined in the *ConfigMap*, will be set in `salt/metalk8s/addons/prometheus-operator/config/prometheus.yaml`.

Custom Alertmanager Configuration

We will use the already existing `metalk8s-alertmanager-config` *ConfigMap* to store the term:*Alertmanager* configuration customizations.

A Salt module will be developed to manipulate this object, so the logic can be kept in only one place.

This module must provide necessary methods to show or edit the configuration in 2 different ways:

- simple
- advanced

The `simple` mode will only display and allow to change some specific configuration, such as the receivers or the inhibit rules, and in an as simple as possible manner for the user.

The `advanced` mode will allow to change all the configuration points, exposing the whole configuration as a plain YAML.

This module will then be exposed through a CLI and a UI.

Retrieve Alert Rules List

To retrieve the list of alert rules, we must use the Prometheus API. This can be achieved using the following route:

```
http://<prometheus-ip>:9090/api/v1/rules
```

This API call should be done in a Salt module `metalk8s_monitoring` which could then be wrapped in a CLI and UI.

Silence an Alert

To silence an alert, we need to send a query to the Alertmanager API. This can be done using the following route:

```
http://<alertmanager-ip>:9093/api/v1/silences
```

With a POST query content formatted as below:

```
{
  "matchers": [
    {
      "name": "alert-name",
      "value": "<alert-name>"
    }
  ],
  "startsAt": "2020-04-10T12:12:12",
  "endsAt": "2020-04-10T13:12:12",
  "createdBy": "<author>",
  "comment": "Maintenance is planned",
  "status": {
    "state": "active"
  }
}
```

We must also be able to retrieve silenced alerts and to remove a silence. This will be done using the API, with the same route using GET and DELETE word respectively:

```
# GET - to list all silences
http://<alertmanager-ip>:9093/api/v1/silences

# DELETE - to delete a specific silence
http://<alertmanager-ip>:9093/api/v1/silence/<silence-id>
```

We will need to provide these functionnalities through a Salt module `metalk8s_monitoring` which could then be wrapped in a CLI and UI.

Extract Rules Tooling

We need to build a tool to extract all alert rules from the *Prometheus* Operator rendered chart `salt/metalk8s/addons/prometheus-operator/deployed/chart.sls`.

Its purpose will be to generate a file (each time this chart is updated) which will then be used to check that what's deployed matches what was expected.

And so, we will be able to see what has been changed when updating *Prometheus* Operator chart and see if there is any change on customizable alert rules.

Rejected Design Choices

Using amtool vs Alertmanager API

Managing alert silences can be done using amtool:

```
# Add
amtool --alertmanager.url=http://localhost:9093 silence add \
  alertname="<alert-name>" --comment 'Maintenance is planned'

# List
amtool --alertmanager.url=http://localhost:9093 silence query

# Delete
amtool --alertmanager.url=http://localhost:9093 silence expire <silence-id>
```

This option has been rejected because, to do so, we need to install an extra dependency (amtool binary) or run the commands inside the *Alertmanager* container, rather than simply send HTTP queries on the API.

Implementation Details

Iteration 1

- Add an internal tool to list all Prometheus alert rules from rendered chart
- Implement Salt formulas to handle configuration customization (`advanced` mode only)
- Provide CLI and UI to wrap the Salt calls
- Customization of node-exporter alert group thresholds
- Document how to:
  - Retrieve the list of alert rules
  - Add a new alert rule
  - Edit an existing alert rule
  - Configure notifications (email, slack and pagerduty)
  - Silence an alert
  - Deactivate an alert

Iteration 2

- Implement the `simple` mode in Salt formulas
- Add the `simple` mode to both CLI and UI
- Update the documentation with the `simple` mode

Documentation

In the Operational Guide:

- Document how to manage silence on alerts (list, create & delete)
- Document how to manage alert rules (list, create, edit)
- Document how to configure alertmanager notifications
- Document how to deactivate an alert
- Add a list of alert rules configured in *Prometheus*, with a brief explanation for each and what can be customized

Test Plan

Add a new test scenario using pytest-bdd framework to ensure the correct behavior of this feature. These tests must be put in the post-merge step in the CI and must include:

- Configuration of a receiver in *Alertmanager*
- Configuration of inhibit rules in *Alertmanager*
- Add a new alert rule in *Prometheus*
- Customize an existing alert rule in *Prometheus*
- Alert silences management (add, list and delete)
- Deployed Prometheus alert rules must match what's expected from a given list (generated by a tool *Extract Rules Tooling*)

3.1.3 Metalk8s predefined Alert rules and Alert Grouping

Context

As part of Metalk8s, we would like to provide the Administrator with built-in rules expressions that can be used to fire alerts and send notifications when one of the High Level entities of the system is degraded or impacted by the degradation of a Low Level component.

As an example, we would like to notify the administrator when the MetalK8s log service is degraded because of some specific observed symptoms:

- not all log service replicas are scheduled
- one of the persistent volumes claimed by one log service replica is getting full.
- Log DB Ingestion rate is near zero

In this specific example, the goal is to invite the administrator to perform manual tasks to avoid having a Log Service interruption in the near future.

Vocabulary

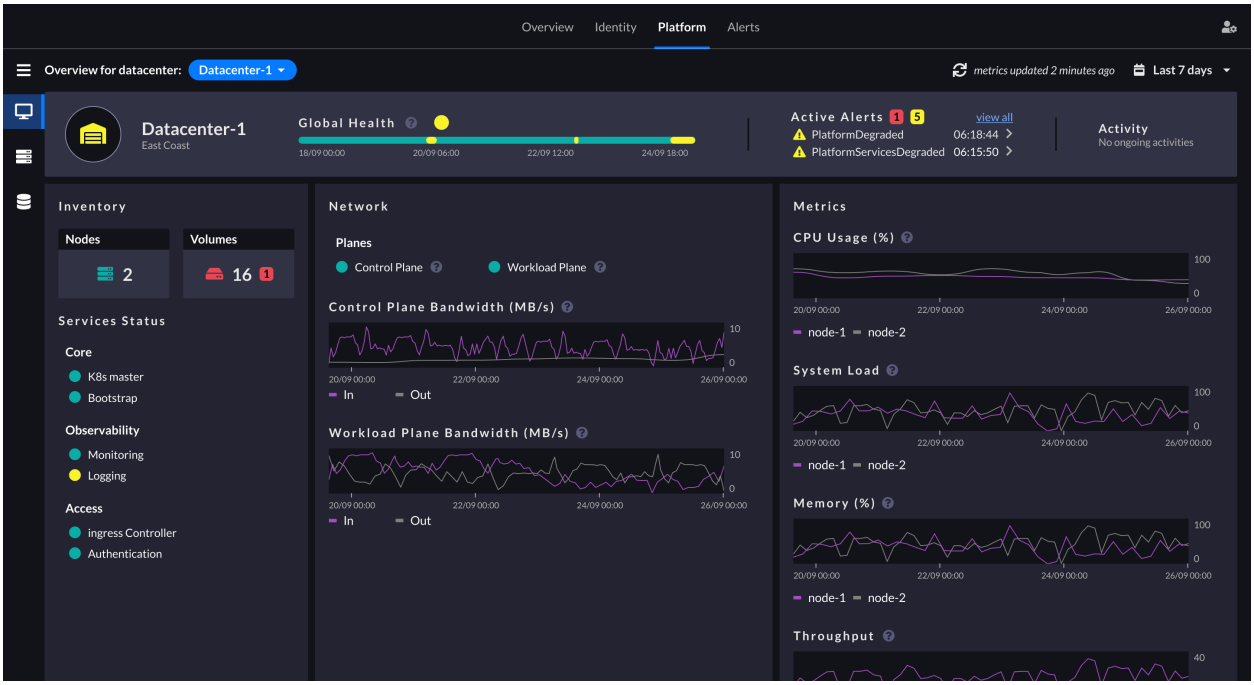
Atomic Alert: An Alert which is based on existing metrics in Prometheus and which is linked to a specific symptom.

High Level Alert: An Alert which is based on other atomic alerts or High Level alerts.

Requirements

When receiving such High Level alerts, we would like the system to guide the administrator to find and understand the root cause of the alert as well as the path to resolve it. Accessing the list of observed low level symptoms will help the administrator's investigation.

Having the High Level Alerts also helps the administrator to have a better understanding of what part/layer/component of the System is currently impacted (without having to build a mental model to guess the impact of any existing atomic alert in the System)

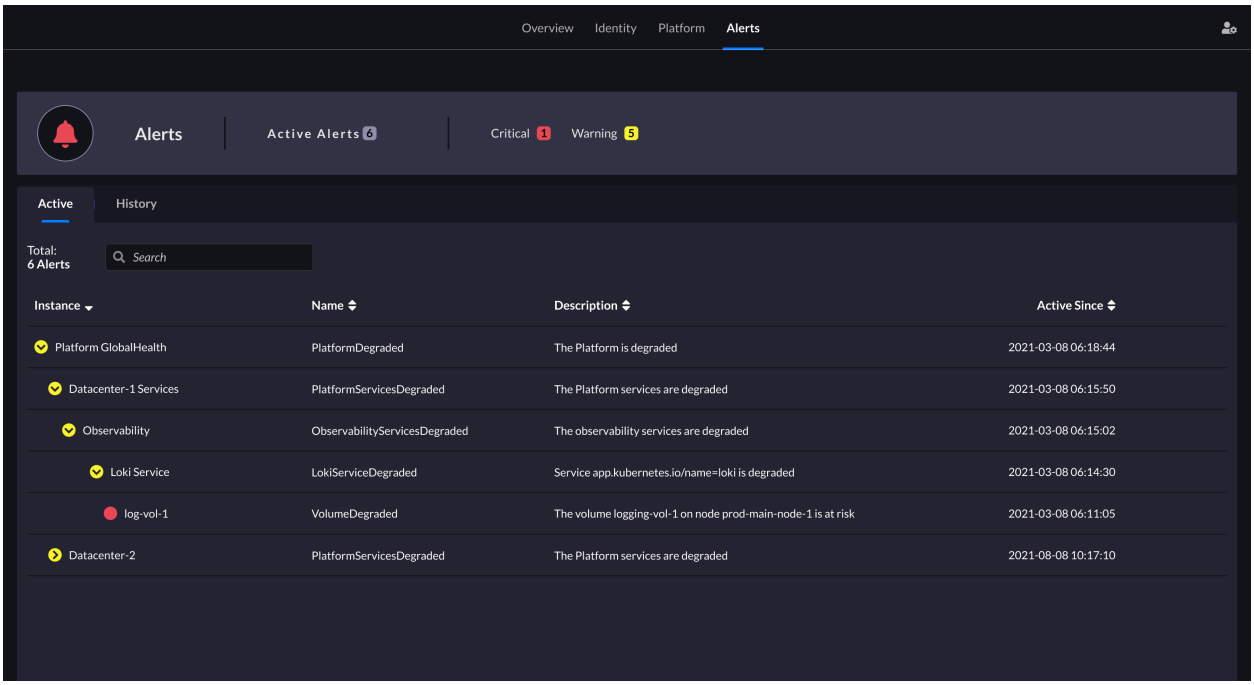


A bunch of atomic alerts are already deployed but we don't yet have the High Level Alerts that we could use to build the above the MetalK8s dashboard. Being able to define the impact of one atomic alert is a way to build those High Level Alerts:

It is impossible to modelize all possible causes through this kind of impacting tree. However, when an alert is received, the system shall suggest other alerts that may be linked to it, (maybe using matching labels).

Also, when accessing the Node or the Volume page / alert tab, the administrator should be able to visualise all the fired alerts that are described under Nodes or Volumes entities.

In the end, the way to visualise the impact of an atomic alert in the alert page is described with the screenshot below:



The High Level alerts should be easily identifiable in order to filter it out in the UI views. Indeed, the a first iteration we might want to display the atomic alerts only until all High Level alerts are implemented and deployed.

Severity Classification

- Critical Alert = Red = Service Offline or At Risk, requires immediate intervention
- Warning Alert = Yellow = Service Degraded, requires planned (within 1 week) intervention
- No Active Alert = Green = Service Healthy

Notifications are either a mail, slack message or whatever routing supported by AlertManager or a decorated icon in the UI.

Data Model

We consider that Nodes and Volumes don't impact the Platform directly. As such they are not belonging to Platform.

Volumes

Nodes

- *System Partitions*

Platform

- *Platform Services*
  - *Core*
  - *Observability*
  - *Access*
- *Network*

Platform

Table 1: PlatformAtRisk

Severity	Critical
Summary	The Platform is at risk
Parent	*none*

Sub Alert	Severity	Filter
PlatformServicesAtRisk	Critical	

Table 2: PlatformDegraded

Severity	Warning
Summary	The Platform is degraded
Parent	*none*

Sub Alert	Severity	Filter
PlatformServicesDegraded	Warning	
ControlPlaneNetworkDegraded	Warning	
WorkloadPlaneNetworkDegraded	Warning	

Nodes

Table 3: NodeAtRisk

Severity	Critical
Summary	Node <nodename> is at risk
Parent	none

Sub Alert	Severity	Filter
KubeletClientCertificateExpiration	Critical	
NodeRAIDDegraded	Critical	
SystemPartitionAtRisk	Critical	

Table 4: NodeDegraded

Severity	Warning
Summary	Node <nodename> is degraded
Parent	none

Sub Alert	Severity	Filter
KubeNodeNotReady	Warning	
KubeNodeReadinessFlapping	Warning	
KubeNodeUnreachable	Warning	
KubeletClientCertificateExpiration	Warning	
KubeletClientCertificateRenewalErrors	Warning	
KubeletPlegDurationHigh	Warning	
KubeletPodStartUpLatencyHigh	Warning	
KubeletServerCertificateExpiration	Warning	
KubeletServerCertificateExpiration	Warning	
KubeletServerCertificateRenewalErrors	Warning	
KubeletTooManyPods	Warning	
NodeClockNotSynchronising	Warning	
NodeClockSkewDetected	Warning	
NodeRAIDDiskFailure	Warning	
NodeTextFileCollectorScrapeError	Warning	
SystemPartitionDegraded	Warning	

Currently no atomic Alert is defined yet for the following

- System Unit (kubelet, containerd, salt-minion, ntp) would need to enrich node exporter
- RAM
- CPU

System Partitions

Table 5: SystemPartitionAtRisk

Severity	Warning
Summary	The partition <mountpoint> on node <nodename> is at risk
Parent	*NodeAtRisk*

Sub Alert	Severity	Filter
NodeFilesystemAlmostOutOfSpace	Critical	
NodeFilesystemAlmostOutOfFiles	Critical	
NodeFilesystemFilesFillingUp	Critical	
NodeFilesystemSpaceFillingUp	Critical	

Table 6: SystemPartitionDegraded

Severity	Warning
Summary	The partition <mountpoint> on node <nodename> is degraded
Parent	*NodeDegraded*

Sub Alert	Sever-ity	Filter
NodeFilesystemAlmostOutOfSpace	Warning	
NodeFilesystemAlmostOutOfFiles	Warning	
NodeFilesystemFilesFillingUp	Warning	
NodeFilesystemSpaceFillingUp	Warning	

Volumes

Table 7: VolumeAtRisk

Severity	Critical
Summary	The volume <volumename> on node <nodename> is at risk
Parent	*multiple parents*

Sub Alert	Sever-ity	Filter
KubePersistentVolumeErrors	Warning	
KubePersistentVolumeFillingUp	Critical	

Table 8: VolumeDegraded

Severity	Warning
Summary	The volume <volumename> on node <nodename> is degraded
Parent	*multiple parents*

Sub Alert	Sever-ity	Filter
KubePersistentVolumeFillingUp	Warning	

Platform Services

Table 9: PlatformServicesAtRisk

Severity	Critical
Summary	The Platform services are at risk
Parent	*PlatformAtRisk*

Sub Alert	Sever-ity	Filter
CoreServicesAtRisk	Critical	
ObservabilityServicesAtRisk	Critical	

Table 10: PlatformServicesDegraded

Severity	Warning
Summary	The Platform services are degraded
Parent	*PlatformDegraded*

Sub Alert	Sever-ity	Filter
CoreServicesDegraded	Warning	
ObservabilityServicesDegraded	Warning	
AccessServicesDegraded	Warning	

Core

Table 11: CoreServicesAtRisk

Severity	Critical
Summary	The Core services are at risk
Parent	*PlatformServicesAtRisk*

Sub Alert	Sever-ity	Filter
K8sMasterServicesAtRisk	Critical	

Table 12: CoreServicesDegraded

Severity	Warning
Summary	The Core services are degraded
Parent	*PlatformServicesDegraded*

Sub Alert	Sever-ity	Filter
K8sMasterServicesDegraded	Critical	
BootstrapServicesDegraded	Critical	

Table 13: K8sMasterServicesAtRisk

Severity	Warning
Summary	The kubernetes master services are at risk
Parent	*CoreServicesAtRisk*

Sub Alert	Severity	Filter
KubeAPIErrorBudgetBurn	Critical	
etcdHighNumberOfFailedGRPCRequests	Critical	
etcdGRPCRequestsSlow	Critical	
etcdHighNumberOfFailedHTTPRequests	Critical	
etcdInsufficientMembers	Critical	
etcdMembersDown	Critical	
etcdNoLeader	Critical	
KubeStateMetricsListErrors	Critical	
KubeStateMetricsWatchErrors	Critical	
KubeAPIDown	Critical	
KubeClientCertificateExpiration	Critical	
KubeClientCertificateExpiration	Critical	
KubeControllerManagerDown	Critical	
KubeletDown	Critical	
KubeSchedulerDown	Critical	

Table 14: K8sMasterServicesDegraded

Severity	Warning
Summary	The kubernetes master services are degraded
Parent	*CoreServicesDegraded*

Sub Alert	Severity	Filter
KubeAPIErrorBudgetBurn	Warning	
etcdHighNumberOfFailedGRPCRequests	Warning	
etcdHTTPRequestsSlow	Warning	
etcdHighCommitDurations	Warning	
etcdHighFsyncDurations	Warning	
etcdHighNumberOfFailedHTTPRequests	Warning	
etcdHighNumberOfFailedProposals	Warning	
etcdHighNumberOfLeaderChanges	Warning	
etcdMemberCommunicationSlow	Warning	
KubeCPUOvercommit	Warning	
KubeCPUQuotaOvercommit	Warning	
KubeMemoryOvercommit	Warning	
KubeMemoryQuotaOvercommit	Warning	
KubeClientCertificateExpiration	Warning	
KubeClientErrors	Warning	
KubeVersionMismatch	Warning	
KubeDeploymentReplicasMismatch	Warning	kube-system/coredns
KubeDeploymentReplicasMismatch	Warning	metalk8s-monitoring/prometheus-adapter
KubeDeploymentReplicasMismatch	Warning	metalk8s-monitoring/prometheus-operator-kube-state-metrics

Table 15: BootstrapServicesDegraded

Severity	Warning
Summary	The bootstrap services are degraded
Parent	*CoreServicesDegraded*

Sub Alert	Severity	Filter
KubePodNotReady	Warning	kube-system/repositories-<bootstrapname>
KubePodNotReady	Warning	kube-system/salt-master-<bootstrapname>
KubeDeploymentReplicasMismatch	Warning	kube-system/storage-operator
KubeDeploymentReplicasMismatch	Warning	metalk8s-ui/metalk8s-ui

Note: The name of the bootstrap node depends on how MetalK8s is deployed. We would need to automatically configure this alert during deployment. We may want to use more deterministic filter to find out the repository and salt-master pods.

Observability

Table 16: ObservabilityServicesAtRisk

Severity	Critical
Summary	The observability services are at risk
Parent	*PlatformServicesAtRisk*

Sub Alert	Severity	Filter
MonitoringServiceAtRisk	Critical	
AlertingServiceAtRisk	Critical	
LoggingServiceAtRisk	Critical	

Table 17: ObservabilityServicesDegraded

Severity	Warning
Summary	The observability services are degraded
Parent	*PlatformServicesDegraded*

Sub Alert	Severity	Filter
MonitoringServiceDegraded	Warning	
AlertingServiceDegraded	Warning	
DashboardingServiceDegraded	Warning	
LoggingServiceDegraded	Warning	

Table 18: MonitoringServiceAtRisk

Severity	Warning
Summary	The monitoring service is at risk
Parent	*ObservabilityServicesAtRisk*

Sub Alert	Severity	Filter
PrometheusRuleFailures	Critical	
PrometheusRemoteWriteBehind	Critical	
PrometheusRemoteStorageFailures	Critical	
PrometheusErrorSendingAlertsToAnyAlertmanager	Critical	
PrometheusBadConfig	Critical	

Table 19: MonitoringServiceDegraded

Severity	Warning
Summary	The monitoring service is degraded
Parent	*ObservabilityServicesDegraded*

Sub Alert	Severity	Filter
VolumeDegraded	Warning	app.kubernetes.io/name=prometheus-operator-prometheus
VolumeAtRisk	Critical	app.kubernetes.io/name=prometheus-operator-prometheus
TargetDown	Warning	To be defined
PrometheusTargetLimitHit	Warning	
PrometheusTSDBReloadsFailing	Warning	
PrometheusTSDBCompactionsFailing	Warning	
PrometheusRemoteWriteDesiredShards	Warning	
PrometheusOutOfOrderTimestamps	Warning	
PrometheusNotificationQueueRunningFull	Warning	
PrometheusNotIngestingSamples	Warning	
PrometheusNotConnectedToAlertmanagers	Warning	
PrometheusMissingRuleEvaluations	Warning	
PrometheusErrorSendingAlertsToSomeAlertmanagers	Warning	
PrometheusDuplicateTimestamps	Warning	
PrometheusOperatorWatchErrors	Warning	
PrometheusOperatorSyncFailed	Warning	
PrometheusOperatorRejectedResources	Warning	
PrometheusOperatorReconcileErrors	Warning	
PrometheusOperatorNotReady	Warning	
PrometheusOperatorNodeLookupErrors	Warning	
PrometheusOperatorListErrors	Warning	
KubeStatefulSetReplicasMismatch	Warning	metalk8s-monitoring/prometheus-prometheus-operator-prometheus
KubeDeploymentReplicasMismatch	Warning	metalk8s-monitoring/prometheus-operator-operator
KubeDaemonSetNotScheduled	Warning	metalk8s-monitoring/prometheus-operator-prometheus-node-exporter

Table 20: LoggingServiceAtRisk

Severity	Critcal
Summary	The logging service is at risk
Parent	*ObservabilityServicesAtRisk*

Sub Alert	Severity	Filter
AlertmanagerConfigInconsistent	Critical	
AlertmanagerMembersInconsistent	Critical	
AlertmanagerFailedReload	Critical	

Table 21: LoggingServiceDegraded

Severity	Warning
Summary	The logging service is degraded
Parent	*ObservabilityServicesDegraded*

Sub Alert	Severity	Filter
VolumeDegraded	Warning	app.kubernetes.io/name=loki
VolumeAtRisk	Critical	app.kubernetes.io/name=loki
TargetDown	Warning	To be defined
KubeStatefulSetReplicasMismatch	Warning	metalk8s-logging/loki
KubeDaemonSetNotScheduled	Warning	metalk8s-logging/fluentbit

Table 22: AlertingServiceAtRisk

Severity	Critcal
Summary	The alerting service is at risk
Parent	*ObservabilityServicesAtRisk*

Sub Alert	Severity	Filter
AlertmanagerConfigInconsistent	Critical	
AlertmanagerMembersInconsistent	Critical	
AlertmanagerFailedReload	Critical	

Table 23: AlertingServiceDegraded

Severity	Warning
Summary	The alerting service is degraded
Parent	*ObservabilityServicesDegraded*

Sub Alert	Severity	Filter
VolumeDegraded	Warning	app.kubernetes.io/name=prometheus-operator-alertmanager
VolumeAtRisk	Critical	app.kubernetes.io/name=prometheus-operator-alertmanager
TargetDown	Warning	To be defined
KubeStatefulSetReplicasMismatch	Warning	metalk8s-monitoring/alertmanager-prometheus-operator-alertmanager
AlertmanagerFailedReload	Warning	

Table 24: DashboardingServiceDegraded

Severity	Warning
Summary	The dashboarding service is degraded
Parent	*ObservabilityServicesDegraded*

Sub Alert	Severity	Filter
KubeStatefulSetReplicasMismatch	Warning	metalk8s-monitoring/prometheus-operator-grafana
TargetDown	Warning	To be defined

Network

Table 25: ControlPlaneNetworkDegraded

Severity	Warning
Summary	The Control Plane Network is degraded
Parent	*PlatformDegraded*

Sub Alert	Severity	Filter
NodeNetworkReceiveErrs	Warning	Need to filter on the proper cp interface
NodeHighNumberConntrackEntriesUsed	Warning	Need to filter on the proper cp interface
NodeNetworkTransmitErrs	Warning	Need to filter on the proper cp interface
NodeNetworkInterfaceFlapping	Warning	Need to filter on the proper cp interface

Table 26: WorkloadPlaneNetworkDegraded

Severity	Warning
Summary	The Workload Plane Network is degraded
Parent	*PlatformDegraded*

Sub Alert	Severity	Filter
NodeNetworkReceiveErrs	Warning	Need to filter on the proper wp interface
NodeHighNumberConntrackEntriesUsed	Warning	Need to filter on the proper wp interface
NodeNetworkTransmitErrs	Warning	Need to filter on the proper wp interface
NodeNetworkInterfaceFlapping	Warning	Need to filter on the proper wp interface

Note: The name of the interface used by Workload Plane and/or Control Plane is not known in advance. As such, we should find a way to automatically configure the Network alerts based on Network configuration.

Note: Currently we don't have any alerts for the Virtual Plane which is provided by kube-proxy, calico-kube-controllers, calico-node. It is not even part of the MetalK8s Dashboard page. We may want to introduce it.

Access

Table 27: AccessServicesDegraded

Severity	Warning
Summary	The Access services are degraded
Parent	*PlatformServicesDegraded*

Sub Alert	Severity	Filter
IngressControllerDegraded	Warning	
AuthenticationDegraded	Warning	

Table 28: IngressControllerDegraded

Severity	Warning
Summary	The Ingress Controllers for CP and WP are degraded
Parent	*AccessServicesDegraded*

Sub Alert	Severity	Filter
KubeDeploymentReplicasMismatch	Warning	metalk8s-ingress/ingress-nginx-defaultbackend
KubeDaemonSetNotScheduled	Warning	metalk8s-system/ingress-nginx-controller
KubeDaemonSetNotScheduled	Warning	metalk8s-system/ingress-nginx-control-plane-controller

Table 29: AuthenticationDegraded

Severity	Warning
Summary	The Authentication service for K8S API is degraded
Parent	*AccessServicesDegraded*

Sub Alert	Severity	Filter
KubeDeploymentReplicasMismatch	Warning	metalk8s-auth/dex

3.1.4 Authentication

Context

Currently, when we deploy MetalK8s we pre-provision a super admin user with a username/password pair. This implies that anyone wanting to use the K8s/Salt APIs needs to authenticate using this single super admin user.

Another way to access the APIs is by using the K8s admin certificate which is stored in `/etc/kubernetes/admin.conf`. We could also manually provision other users, their corresponding credentials as well as role bindings but this current approach is inflexible to operate in production setups and security is not guaranteed since username/password pairs are stored in cleartext.

We would at least like to be able to add different users with different credentials and ideally integrate K8s authentication system with an external identity provider.

Managing K8s role binding between user/groups high-level roles and K8s roles is not part of this specification.

Requirements

Basically, we are talking about:

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

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

User Stories

Pre-provisioned user and password change

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

User Management with local IdP

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

- Cluster Admin
- Solution Admin
- Read Only

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

External IDP Integration

As an IT Generalist, I want to leverage my organisation's IDP to reuse already provisioned users & groups. The way we do that integration is through a CLI tool which does not require to have deep knowledge in K8s or in any local IDP specifics. When External IDP Integration is set up, we can always use local IDP to authenticate.

Authentication check

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

Configuration persistence

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

SSO between Admin UIs

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

Open questions

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

Design Choices

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

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

Rejected design choices

Static password file Vs OpenID Connect

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

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

X.509 certificates Vs OpenID Connect

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

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

Keycloak Vs Dex

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

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

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

Why not Keycloak?

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

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

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

Unexploited choices

- Guard

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

- ORY Hydra

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

Implementation Details

Iteration 1

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

Iteration 2

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

Iteration 3

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

Iteration 4

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

The following operations will be supported using the CLI tool:

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

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

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

Iteration 5

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

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

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

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

Documentation

In the Operational Guide:

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

In the Installation/Quickstart Guide

- Document how to setup/change the superuser password

Test Plan

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

3.1.5 Centralized CLI

Context

MetalK8s comes with a set of services to operate and monitor the K8s cluster. All operations that need to be performed by the Platform Administrator could be categorized as follow:

- Cluster Resources Administration (Nodes, Volumes, Deployments, ...)
- Cluster Administration (Install, Upgrade, Downgrade, Backup, Restore, ...)
- Solution Administration (CRUD Environment, Import/Remove Solution, ...)
- Cluster Service Administration (Configure Dex, Prometheus, Alert Manager, ...)

K8s provides the kubectl CLI, enabling all kind of interactions with all Kubernetes resources, through k8s apiserver, but its usage often requires to build verbose YAML files. Also it does not leverage everything MetalK8s exposes through the salt API. It is shipped as an independent package and can be deployed and run from anywhere, on any OS.

Currently, MetalK8s provides other set of scripts or manual procedures, but those are located in various locations, their usage may vary and they are not developed using the same logic.

This makes the CLI and associated documentation not super intuitive and it also makes the maintenance more expensive in the long term.

The goal of the project is to provide MetalK8s administrator with an intuitive and easy to use set of tools in order to administrate and operate a finite set of functionalities.

Because kubectl is already in place and is well known by Kubernetes administrators, it will be used as a standard to follow, as much as possible, for all other MetalK8s CLIs:

- CLI provides an exhaustive help, per action, with relevant examples
- CLI provides <action> help when the command is not valid
- CLI is not interactive (except if password input is needed)
- CLI should not require password input
- CLI provides a dryrun mode for intrusive operations
- CLI provides a verbose (or debug) mode

- CLI implementation relies on secure APIs
- CLI support action completion for easy discovery
- CLI output is standardized and human readable by default
- CLI output can be formatted in JSON or YAML

When it is possible, it would make sense to leverage kubectl plugin

Most functionalities are exposed through 2 distinct CLI:

- kubectl: enriched with metalk8s plugin, to interact with both k8s apiserver and salt API, and that can be executed from outside of the cluster.
- metalk8sctl: a new CLI, exposing specific MetalK8s functionalities, that are not interacting with k8s apiserver, and that must be executed on cluster node host.

Some cluster configurations will be achievable through documented procedures, such as changing one cluster server hostname.

Other specific solution kubectl plugin may also be provided by a solution.

To know which command must be used, administrator will rely on MetalK8s documentation. Documentation will be updated accordingly.

In order to operate the cluster with kubectl plugins from outside of the cluster, plugin binary will be available for download from the bootstrap node or from MetalK8s release repository. The metalk8sctl and kubectl are deployed and available by default on bootstrap nodes.

Requirements

Not listing all commands that are already available through kubectl. Only describing commands that are missing or commands that can be simplified using new command line arguments.

Cluster Resources Administration

tool: kubectl metalk8s

action	resource type	resource id	parameters
create	node	name	ssh-user, hostname or ip, ssh port ssh-key-path, sudo-required, roles
deploy	node	name…	<dry-run>
create	volume	name	type, nodeName, storageClassName, <devicePath>, <size>, <labels>

Cluster Administration

tool: metalk8sctl

Resource	action	parameters
bootstrap	deploy	
archive	import	path_to_iso
archive	get	<name>
archive	delete	path_to_iso or path_to_mountpoint or name
cluster	upgrade	dest-version, <dry-run>
cluster	downgrade	dest-version, <dry-run>
etcd	health	
bootstrap	backup	
bootstrap	restore	backup-file

Solution Administration

Note: Import and unimport of solution are done the same way as MetalK8s `archive` using `metalk8sctl archive import ...`.

---

tool: metalk8sctl

Resource	action	parameters
solution	activate	name, version
solution	deactivate	name
solution	get	<name>, <version>

tool: kubectl metalk8s

action	resource type	parameters
create	environment	name, <description>, <namespace>
delete	environment	name, <namespace>
get	environment	<name>
add	solution	name, version, environment, <namespace>
delete	solution	name, environment, <namespace>
get	solution	<name>, environment

Cluster Service Administration

tool: kubectl metalk8s

action	resource type	resource id	parameters
The following edit commands are doing both configuration update and applying the configuration.			
edit	grafana-config	name	open an editor
edit	am-config	name	open an editor
edit	prom-config	name	open an editor
edit	dex-config	name	open an editor

Design Choices

Two distinct CLI:

- a `metalk8s` kubectl plugin with subcommands to interact with Kubernetes API, and Salt API if needed.
- a `metalk8sctl` CLI with subcommands for action that need to interact with the local machine, but may also interact with Kubernetes API and Salt API if needed.

`metalk8s` kubectl plugin

Language

Go is chosen as the language for kubectl plugin for the following reasons:

- Great interactions with Kubernetes API.
- Often used for operators and kubectl plugins (Sample CLI plugin, Helpers for kubectl plugins).
- Easy to ship because it's a statically compiled binary, no deps to provide.
- Simple deployment (no real requirements), just drop a binary in the PATH.

Input and Output

Each command should follow the kubectl style and standard as much as possible:

- Command style:

```
kubectl metalk8s <action> <resource>
```

- Interactive:

  No interaction with the user, except when it's an `edit` command an editor is opened (if needed) and when a password is required a prompt appears to ask it.

- Output style:

  Default human-readable output (`<object> <action>ed`).

  A `--output`, `-o` option to change output format, at least support for `json` and `yaml` (`jsonpath` and `go-template` when it's possible).

  Internally each action result should be an "object" (e.g.: single-level dictionary) containing several informations, at least:

  - name
  - message
  - result (True or False)
  - an elapsed time (to know each action time)

  A `--verbose`, `-v` option to change log level verbosity (default output to `stderr`), using Kubernetes log library.

  By default each command will wait for a result but, when it's possible, a `--async` option should allow to do not wait for a result and just output an ID (e.g.: Job ID for Salt) that can be used to watch for the result.

SaltAPI interaction

If the plugin needs to access Salt API then it should use the service proxy `http://<apiserver_host>/api/v1/namespaces/kube-system/services/https:salt-master:api/proxy/`.

For each and every Salt API call plugin will need authentication on apiserver to access the Salt API service proxy and also to Salt API.

Note: Right now, Salt API only accepts authentication using Bearer token, but in kubeconfig we could have certificates authentication so this kind of kubeconfig will not work with this kubectl plugin.

Add support for certificates based authentication in Salt API look quite hard and costly.

Deployment

Plugin should be developed as one single binary `kubectl-metalk8s` available from the ISO, easy buildable from GitHub repository and also as a System Package for Operating System supported by MetalK8s.

The package should install the plugin in `/usr/bin` directory by default.

This package should be installed on the bootstrap node by default.

Rejected design choice

- `Bash` kubectl plugin: Bash is great to do simple actions but not when you need to do interaction with some API like Kubernetes API or Salt API.
- `Python` kubectl plugin: Python allows us to do complicated actions and great interactions with APIs but interactions between `Go` and Kubernetes are much easier, given the large number of example available.

`metalk8sctl` CLI

Language

`Python` is chosen as the language for `metalk8sctl` for the following reasons:

- Ability to interact with Salt Python client API.
- Python installation needed anyway by Salt-minion.

Note: Python version 3 will be used as version 2 is end of life since beginning of 2020.

Input and Output

- Command style:

```
metalk8sctl <resource> <action>
```

- Interactive:

  Never.

- Output style:

  Human readable output, do not necessarily need for "machine output" like JSON and YAML.

  The output should display useful information from Salt returns when needed, and in case of error, only show relevant error message(s) from Salt.

Salt interaction

All Salt interaction should be done using Salt Python client API and not use the `salt-call`, `salt`, `salt-run` binary at all.

This Salt Python client API allows us to interact with Salt-master directly from the host machine as Python API directly acts on the Salt sockets and does not need to execute a command inside the Salt-master container.

Deployment

`metalk8sctl` should be available from the ISO and also as a System Package for Operating System supported by MetalK8s.

This package should be installed on the bootstrap node automatically after a fresh install.

As this CLI is used to do the first bootstrap deployment we will need another script (likely `bash`) to configure local repositories and install `metalk8sctl` package with all dependencies.

Note: This CLI cannot run from outside of the cluster and need to have root access on the machine to run.

That's why this CLI do not need any specific authentication on the cluster itself, interaction with all machines will be done using Salt.

Rejected design choice

- `bash` MetalK8s CLI: Bash is great to do simple actions but not when you need to do interaction with Salt, Salt API, and Kubernetes API.
- Do not follow kubectl style for the command (`<action> <resource>`), it does not make sense to regroup command per action as actions are really different and this CLI only manages a few resources.

Implementation Details

Two different projects that can be started in parallel.

First have a simple framework to implement a simple command, then each command would update the framework if needed.

Check *Requirements* for a full list of commands.

Documentation

All command should be documented in the Operational Guide with a reference to it when it's needed in the Installation Guide.

All commands and sub-commands should have a `--help` option to explain a bit of usage of this specific command and available options.

Test plan

For `metalk8sctl`:

- Add unit tests for internal functions using Pytest
- Most of the command are already used during functional test (some may need to be added in PyTest BDD)

For `metalk8s` kubectl plugin:

- Add unit tests for internal functions using Golang testing framework
- Add functional test for all plugin commands in PyTest BDD

3.1.6 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

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

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

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, ...)

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.

3.1.7 Cluster and Services Configurations and Persistence

Context

MetalK8s comes with a set of tools and services that may need to be configured on site. At the same time, we don't want the administrator of the cluster to master each and every service of the cluster. We also don't want to allow all kind of configurations since it would make the system even more complex to test and maintain over time.

In addition to those services, MetalK8s deployment may have to be adapted depending on the architecture of the platform or depending on the different use cases and applications running on top of it.

It can be:

- The BootstrapConfig,
- The various roles and taints we set on the node objects of the cluster
- The configurations associated to solutions, such as the list of available solutions, the environments and namespaces created for a solution

Be it services or MetalK8s configurations, we need to ensure it is persisted and resilient to various type of events such as node reboot, upgrade, downgrade, backup, restore.

Requirements

User Stories

Available Settings

As a cluster administrator, I have access to a finite list of settings I can customize on-site in order to match with my environment specificities:

- List of static users and credentials configured in Dex
- Integration with an external IDP configuration in Dex
- Existing Prometheus rules edition and new rules addition
- Alert notifications configuration in Alert Manager
- New Grafana dashboards or new Grafana datasources
- Number of replicas for the Prometheus, Alert Manager, Grafana or Dex deployments
- Changing the path on which the MetalK8s UI is deployed
- Modifying OIDC provider, client ID or scopes
- Adding custom menu entries

Note: Other items will appear as we add new configurable features in MetalK8s

---

Settings Documentation

As a cluster administrator, I have access to a documented list of settings I can configure in the Operational Guide.

Persistence of Configurations

As a cluster administrator, I can upgrade or downgrade my cluster without losing any of the customised settings described above.

Backup and Restoration

As a cluster administrator, when I am doing a backup of my cluster, I backup all the customised settings described above and I can restore it when restoring the MetalK8s cluster or I can re apply part or all of it on a fresh new cluster.

Expert-mode Access

As a MetalK8s expert, I can use `kubectl` command(s) in order to edit all settings that are exposed. The intent is to have a method / API that an expert could use, if the right CLI tool or GUI is not available or not functioning as expected.

Design Choices

ConfigMap is chosen as a unified data access and storage media for cluster and service configurations in a MetalK8s cluster based on the above requirements for the following reasons:

- Ability to support Update operations on ConfigMaps with CLI and UI easily using our already existing python kubernetes module.
- Guarantee of adaptability and ease of changing the design and implementation in cases where customer needs evolve rapidly.
- ConfigMaps are stored in the *etcd* database which is generally being backed up. This ensures that user settings cannot be lost easily.

How it works

During Bootstrap, Upgrade or Downgrade stages, when we are assertive that the K8s cluster is fully ready and available we could perform the following actions:

- Firstly, create and deploy ConfigMaps that will hold customizable cluster and service configurations. These ConfigMaps should define an empty *config.yaml* in the data section of the ConfigMap for later use.

  A standard layout for each customizable field could be added in the documentation to assist MetalK8s administrator in adding and modifying customizations.

  To simplify the customizing efforts required from MetalK8s administrators, each customizable ConfigMap will include an example section with inline documented directives that highlight how users should add, edit and remove customizations.

- In an Addon config file for example; *salt/metalk8s/addons/prometheus-operator/config/alertmanager.yaml*, define the keys and values for default service configurations in a YAML structured format.
  - The layout of service configurations within this file could follow the format:

```
# Configuration of the Alertmanager service
apiVersion: addons.metalk8s.scality.com/v1alpha1
kind: AlertmanagerConfig
spec:
  # Configure the Alertmanager Deployment
  deployment:
    replicas: 1
```

- During Addon manifest rendering, call a Salt module that will merge the configurations defined within the customizable ConfigMap to those defined in *alertmanager.yaml* using a Salt merge strategy.

  Amongst other merge technique such as *aggregate*, *overwrite*, *list*, the *recurse* merge technique is chosen to merge the two data structures because it allows deep merging of python dict objects while being able to support the aggregation of list structures within the python object.

  Aggregating list structures is particularly useful when merging the pre-provisioned Dex static users found in the default configurations to those newly defined by Administrators especially during upgrade. Without support for list merge, pre-provisioned Dex static users will be overwritten during merge time.

Recurse merge strategy example:

Merging the following structures using *salt.utils.dictupdate.merge*:

- Object (a) (MetalK8s defaults):

```
apiVersion: addons.metalk8s.scality.com/v1alpha1
kind: AlertmanagerConfig
spec:
  deployment:
    replicas: 1
```

- Object (b) (User-defined configurations from ConfigMap):

```
apiVersion: addons.metalk8s.scality.com/v1alpha1
kind: AlertmanagerConfig
spec:
  deployment:
    replicas: 2
  notification:
    config:
      global:
        resolve_timeout: 5m
```

- Result of Salt *recurse* merge:

```
apiVersion: addons.metalk8s.scality.com/v1alpha1
kind: AlertmanagerConfig
spec:
  deployment:
    replicas: 2
  notification:
    config:
      global:
        resolve_timeout: 5m
```

The resulting configuration (a python object) will be used to populate the desired configuration fields within each Addon chart at render time.

The above approach is flexible and fault tolerant because in a MetalK8s cluster, once the user-defined ConfigMaps are absent or empty during Addon deployment, merging will yield no changes and we can effectively use default values packaged alongside each MetalK8s Addon to run the deployment.

Using Salt states

Once a ConfigMap is updated by the user (say a user changes the number of replicas for Prometheus deployments to a new value), then perform the following actions:

- Apply a Salt state that reads the ConfigMap object, validates the schema and checks the new values passed and re-applies this configuration value to the deployment in question.
- Restart the Kubernetes deployment to pickup newly applied service configurations.

Storage format

A YAML (K8s-like) format was chosen to represent the data field instead of a flat key-value structure for the following reasons:

- YAML formatted configurations are easy to write and understand hence it will be simpler for users to edit configurations.
- The YAML format benefits from bearing a schema version, which can be checked and validated against a version we deploy.
- YAML is a format for describing hierarchical data structures, while using a flat key-value format would require a form of encoding (and then, decoding) of this hierarchical structure.

A sample ConfigMap can be defined with the following fields.

```
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: <namespace>
  name: <config-name>
data:
  config.yaml: |-
    apiVersion: <object-version>
    kind: <kind>
    spec:
      <key>: <values>
```

Use case 1:

Configure and store the number of replicas for service specific Deployments found in the *metalk8s-monitoring* namespace using the ConfigMap format.

```
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metalk8s-monitoring
  name: metalk8s-grafana-config
data:
  config.yaml: |-
    apiVersion: metalk8s.scality.com/v1alpha1
    kind: GrafanaConfig
    spec:
      deployment:
        replicas: 2
```

Non-goals

This section contains requirements stated above which the current design choice does not cater for and will be addressed later:

- Persisting newly added Grafana dashboards or new Grafana datasources especially for modifications added via the Grafana UI cannot be stored in ConfigMaps and hence will be catered for later.
- As stated in the requirements, adding and editing Prometheus alert rules is also not covered by the chosen design choice and will need to be addressed differently. Even if we could use ConfigMaps for Prometheus rules, we prefer relying on the Prometheus Operator and it's CRD (PrometheusRule).

Rejected design choices

Consul KV vs ConfigMap

This approach offers a full fledge KV store with a /kv endpoint which allows CRUD operations on all KV data stored in it. Consul KV also allows access to past versions of objects and has an optimistic concurrency when manipulating multiple objects.

Note that, Consul KV store was rejected because managing operations such as performing full backups, system restores for a full fledged KV system requires time and much more efforts than the ConfigMap approach.

Operator (Custom Controller) vs Salt

Operators are useful in that, they provide self-healing functionalities on a reactive basis. When a user changes a given configuration, it is easy to reconcile and apply these changes to the in-cluster objects.

The Operator approach was rejected because it is much more complex, requires much more effort to realize and there is no real need for applying changes using this method because configuration changes are not frequent (for a typical MetalK8s admin, changing the number of replicas for a given deployment could happen once in 3 months or less) as such, having an operator watch for object changes is not significant and not very useful at this point in time.

In the Salt approach, Salt Formulas are designed to be idempotent ensuring that service configuration changes can be applied each time a new configuration is introduced.

Implementation Details

Iteration 1

- Define and deploy new ConfigMap stores that will hold cluster and service configurations as listed in the requirements. For each ConfigMap, define its schema, its default values, and how it impacts the configured services
- Template and render Deployment and Pod manifests that will make use of this persisted cluster and service configurations
- Document how to change cluster and service configurations using kubectl
- Document the entire list of configurations which can be changed by the user

Iteration 2

- Provide a CLI tool for changing any of the cluster and service configurations:
  - Count of replicas for chosen Deployments (Prometheus)
  - Updating a Dex authentication connector (OpenLDAP, AD and staticUser store)
  - Updating the Alertmanager notification configuration
- Provide a UI interface for adding, updating and deleting service specific configurations for example Dex-LDAP connector integration.
- Provide a UI interface for listing MetalK8s available/supported Dex authentication Connectors
- Provide a UI interface for enabling or disabling Dex authentication connectors (LDAP, Active Directory, StaticUser store)
- Add a UI interface for listing Alertmanager notification systems MetalK8s will support (Slack, email)
- Provide a UI interface for adding, modifying and deleting Alertmanager configurations from the listing above

Documentation

In the Operational Guide:

- Document how to customize or change any given service settings using the CLI tool
- Document how to customize or change any given service settings using the UI interface
- Document the list of service settings which can be configured by the user
- Document the default service configurations files which are deployed along side MetalK8s addons

Test Plan

- Add test that ensures that update operations on user configurations are propagated down to the various services
- Add test that ensures that after a MetalK8s upgrade, we do not lose previous customizations.
- Other corner cases that require testing to reduce error prone setups include:
  - Checking for invalid values in a user defined configuration (e.g setting the number of replicas to a string ("two"))
  - Checking for invalid formats in a user configuration
- Add tests to ensure we could merge a service configuration at render time while keeping user-defined modifications intact

3.1.8 Control Plane Ingress

Context

Initially, Control Plane Ingress Controller was deployed using a DaemonSet with a ClusterIP service using Bootstrap Control Plane IP as External IP and then configuring all Control Plane components using this "External IP" (like OIDC and various UIs).

So it means, we can only reach those components using Bootstrap Control Plane IP, and if, for whatever reason, Bootstrap is temporary down you can no longer access any UIs and you need to reconfigure all components manually, change

the IP used everywhere to another Control Plane node IP, in order to restore access to Control Plane components, or if the Bootstrap is down permanently you need to restore the Bootstrap node.

Here, we want to solve this issue and make Control Plane components Highly Available, so if you lose one node, including Bootstrap Node, you can still access various UIs. NOTE: In this document, we do not talk about the High Availability of the components itself but really only to the access through the Ingress (e.g.: We do not want to solve salt-master HA here).

User Stories

MetalK8s and Grafana UIs HA

I have a multi-node MetalK8s cluster, with at least 3 Control Plane nodes if I lose one of the Control Plane nodes (including the Bootstrap one) I can still access and authenticate on the MetalK8s and Grafana UIs.

Design Choices

To have a proper HA Control Plane Ingress we want to use a Virtual IP using MetalLB so that we can rely on layer2 ARP requests when possible.

But in some network, it may be not possible to use this method so we also let the possibility to not use MetalLB but instead just assign an External IP, provided by the user, that we expect to be a Virtual IP, and we do not manage on our side but it's managed by the user using whatever mechanism to switch this IP between Control Plane nodes.

To summarize 2 different deployments possible depending on the user environment:

- Using VIP managed by Layer2 ARP with MetalLB (builtin MetalK8s)
- Using a user-provided IP that should switch between Control Plane nodes (managed by the user)

NOTE: In those 2 approaches we want the user to provide the Control Plane Ingress IP he wants to use.

Rejected Design Choices

Manage Virtual IP by MetalK8s

Instead of using MetalLB to manage the Virtual IP we could have manage this Virtual IP directly in MetalK8s with KeepAliveD (or any other "HA tool") this way we really controle the Virtual IP.

This approach was rejected because it seems to do not provide any real advantage compare to use MetalLB directly that will manage this Virtual IP for us and may provide a bunch of other useful feature for the future.

Rely on DNS resolution

Instead of using a Virtual IP we could rely on DNS resolution use this FQDN to configure Control Plane components. In this case we let the user configure his DNS to resolve on an IP of one Control Plane node.

WIth this approach we let the DNS server handle the "High Availability" of the Control Plane UIs, basically if we lose the node that resolve the DNS then we expect the DNS server to switch to another IP of a working Control Plane node.

This approach was rejected because it's not a real HA as we expect DNS server to have some intelligence which it likely not the case in most of user environments.

Use Nodes Control Plane IPs

Instead of using a Virtual IP or a FQDN we could still rely on Control Plane IPs and configure all Control Plane components using either relative path either all node Control Plane IPs. So that we can reach all Control Plane components using any Control Plane IP.

With this approach we do not need any specific infrastructure on the user environment but it means every time we add a new Control Plane node we need to re-configure every Control Plane component that use Control Plane IPs but we also need to re-configure every Control Plane component when we remove a Control Plane node.

This approach was rejected because Control Plane components we deploy today in MetalK8s does not seems to all support relative path and it does not seems to be possible to have proper HA without re-configuring some Control Plane components when we lose a Control Plane node.

Non-Goals

These points may be addressed later (or not) but in this document, we focus on a first simple deployment that (should) fit in most of the user environments.

- Manage Workload Plane Ingress HA
- Manage BGP routers

Implementation Details

Control Plane Ingress IP

In order to configure all Control Plane components we need a single IP as Control Plane Ingress, so we expect the user to provide this Control Plane Ingress IP in the Bootstrap configuration file. To have some backward compatibility this Ingress IP is only mandatory when you use MetalLB and will default to Bootstrap Control Plane IP if not (so that we have the same behavior as before).

This Control Plane Ingress IP can be changed at any time just by editing the Bootstrap configuration file and follow a simple documented procedure with some Salt states to reconfigure every component that needs to be re-configured.

NOTE: Changing this Control Plane Ingress IP means we need to reconfigure all Kubernetes APIServer since we use this Ingress IP as an OIDC provider.

MetalLB Configuration

MetalLB is not deployed in every environment so it needs to be enabled from the Bootstrap configuration file, that's why we have a new field about MetalLB in the Control Plane network section.

Today, we only allow MetalLB using Layer2 so we do not need to make MetalLB configuration configurable, if MetalLB is enabled in Bootstrap configuration MetalK8s will deploy the following configuration for MetalLB:

```
address-pools:
- name: control-plane-ingress-ip
  protocol: layer2
  addresses:
  - <control-plane ingress ip>/32
  auto-assign: false
```

Same as the Control Plane Ingress IP, we can switch from non-MetalLB to MetalLB (and the opposite) at any time just by following the same procedure.

Deployment

As for every other addon in MetalK8s, we will use the MetalLB helm chart and render this one using a specific "option" file. But this one will not be always deployed as we only want to deploy it when a specific key is set in the Bootstrap configuration file, so in the Salt pillar at the end.

When we use MetalLB we do not want to use the same NGINX Ingress Controller deployments, since MetalLB will be the entry point in the Kubernetes cluster we do not need to use a DaemonSet running on every Control Plane nodes, instead, we will use a Deployment with 2 replicas.

We also need to configure the Service for Ingress Controller differently depending on if we use MetalLB or not when we use it we want to use a LoadBalancer service, set the LoadBalancerIP to IngressIP provided by the user and set externalTrafficPolicy to Local. If we do not use MetalLB then we use ClusterIP Service with IngressIP provided by the user as External IPs.

It means the deployment of NGINX Ingress Controller depends on some Salt pillar values, also since we want to be able to switch between MetalLB and non-MetalLB we need to make sure the Salt states that deploy NGINX Ingress Controller remove no-longer-needed objects (e.g.: if you switch from non-MetalLB to MetalLB you want to remove the DaemonSet for NGINX Ingress Controller).

Documentation

- Describe all new Bootstrap configuration fields
- Add a simple procedure to change the Control Plane Ingress IP and reconfigure all Control Plane conponents that need to.

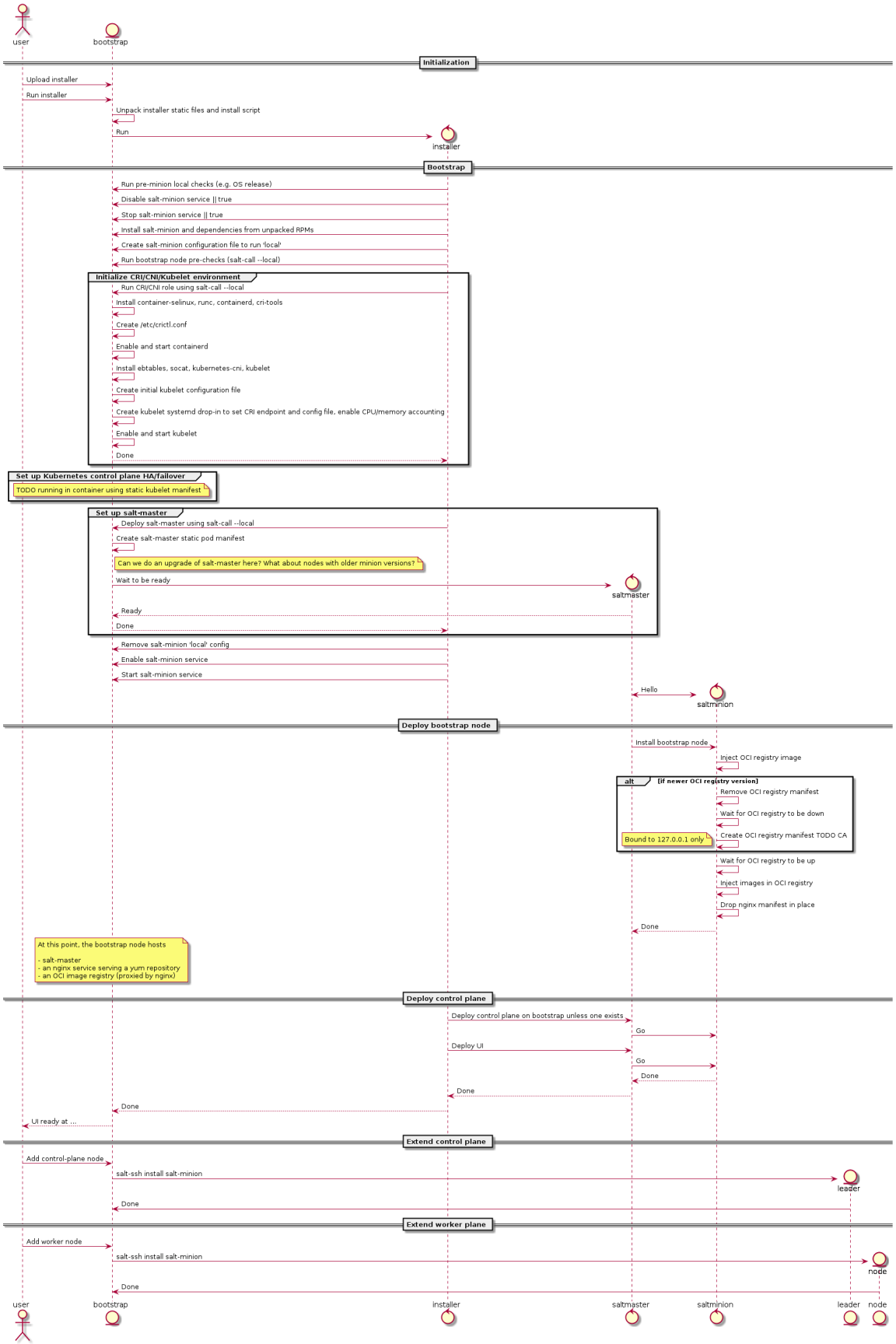
Test Plan

Add some End-to-End tests in the CI:

- Use MetalLB and a VIP as Control Plane Ingress IP
- Test failover of MetalLB VIP
- Change Control Plane Ingress IP using documented procedure
- Switch from non-MetalLB to MetalLB using documented procedure

3.1.9 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

Some notes

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

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

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

Configuration

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

- CIDR (i.e. `x.y.z.w/n`) of the control plane networks to use

  Given these CIDR, we can find the address on which to bind services like `etcd`, `kube-apiserver`, `kubelet`, `salt-master` and others.

  These should be existing networks in the infrastructure to which all hosts are connected.

  This is a list of CIDRs, which will be tried one after another, to find a matching local interface (i.e. hosts comprising the cluster may reside in different subnets, e.g. control plane in VMs, workload plane on physical infrastructure).

- CIDRs (i.e. `x.y.z.w/n`) of the workload plane networks to use

  Given these CIDRs, we can find the address to be used by the CNI overlay network (i.e. Calico) for inter-`Pod` routing.

  This can be the same as the control plane network.

- CIDR (i.e. `x.y.z.w/n`) of the `Pod` overlay network

  Used to configure the Calico `IPPool`. This must be a non-existing network in the infrastructure.

  Default: `10.233.0.0/16`

- CIDR (i.e. `x.y.z.w/n`) of the `Service` network

  Default: `10.96.0.0/12`

Firewall

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

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

These services include:

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

3.1.10 Interaction with Kubernetes API from Salt

Context

In MetalK8s we need to interact with Kubernetes API from Salt in order to create and edit various objects. Since the Salt Kubernetes module is not flexible enough (at least today) we decided to build our own Salt module that will cover all we need in MetalK8s. This Salt module relies on python-kubernetes lib so that we have a proper python class for every Kubernetes object and we can validate the object content without any call to the actual Kubernetes API.

But python-kubernetes lib is a bit late on Kubernetes versions (at the time of writing, the latest python-kubernetes is for k8s-1.18 and k8s 1.22.1 just came out) so it means that some Kubernetes objects from the latest versions cannot be managed by this python lib.

Here, we want to solve this issue and make sure when a new Kubernetes version comes out that we can easily support interaction with all the Kubernetes API from Salt.

Design Choices

Instead of relying on model and API object from python-kubernetes, which is the representation from `openapi` of a specific Kubernetes version, we can directly use python-base which is what is used in the back by python-kubernetes.

This python-base has a `DynamicClient` that can be used to manage any Kubernetes objects no matter the Kubernetes version. It's really more flexible it's mean that we do not validate the manifest we send to Kubernetes API, but the manifest can be validated using kubernetes-validate lib.

This kubernetes-validate lib can have the same issue as python-kubernetes as it needs to be generated for every Kubernetes version, but it seems a bit more up-to-date and this validation is not strictly needed to create the objects, so if a specific version of kubernetes-validate is not yet released we could just remove the validation waiting for a new version to be released.

In order to validate the request or test it we can use `dryRun` parameter from Kubernetes API, it could be really usefull especially when using `test` mode from Salt.

Rejected Design Choices

Migrate to `kubectl`

To have proper support of the full Kubernetes API we could use directly `kubectl`, which is the CLI build as part of Kubernetes.

The advantage of using `kubectl` is that we do not have anything to do in order to support interaction with the new Kubernetes API and nothing (or almost) to maintain in our Salt module that uses this CLI.

This approach was rejected because we need to use a CLI in order to do some "HTTP query" to Kubernetes API that's a bit odd and it add an extra layer that does not bring a lot of value compare to selected design.

Contribute to python-kubernetes Lib

We could contribute to python-kubernetes in order to make it compatible with the latest versions of Kubernetes.

This approach was rejected, for the moment, because python-kubernetes start to be really late on Kubernetes versions, and this work, likely, needs to be done for every new Kubernetes version.

Direct HTTP Query to Kubernetes API from Python

We could build a Salt module that directly interacts with Kubernetes API the way we want so that we do not rely on python-kubernetes but only implement the interactions we need.

This approach was rejected because it means this Salt module needs to be maintained for every new Kubernetes version and for every API/object we want to support.

Implementation Details

First Iteration

In order to be able to use `DynamicClient` from python-base we still need to install a python-kubernetes version since python-base is not packaged, but we do not need this python-kubernetes version to be "compatible" with the Kubernetes version we run since we rely on `DynamicClient` that should be compatible with all Kubernetes versions.

Rework all Salt execution modules that interact with Kubernetes so that it simply uses `DynamicClient`, some examples can be found here.

The Salt state module will likely not be changed and will have the same function as today (object_present, object_absent, object_updated) so it means most of the Salt SLS will not be changed as well.

The Salt execution module will change a bit but the function exposed will likely stay the same.

Most of the logic from the Kubernetes salt utils module can be removed.

Second Iteration (if needed)

If needed we can add some validation.

We can validate the manifest content before sending it to Kubernetes API, to do so, we need to install kubernetes-validate python lib in the Salt master container, and then it can be used easily directly from a `DynamicClient` instance using `validate` function.

Use `dryRun` parameter from Kubernetes API so that we can validate the request.

Documentation

Normally nothing to change.

Test Plan

Update the Salt unit tests for the execution module.

Existing end to end tests should be sufficient.

3.1.11 Log Centralization

Context

MetalK8s value is to provide, out of the box, some services in order to ease the monitoring and operation of the platform as well as workloads running on top of it.

It currently provides monitoring service, powered by Prometheus and AlertManager, in order to expose metrics and alerts for both control-plane and workload-plane components as well as for cluster nodes HW and OS.

The logs generated by the platform and the workloads constitute an essential piece of information when it comes to understanding the root cause of a failure or a performance degradation. Because of the distributed nature of MetalK8s and workloads running on top of it, the administrators need some tooling to ease the analysis of near real time and past logs, from a central endpoint. As such these logs should be stored on the platform, for a configurable period. Browsing the logs is accessible through an API and a UI. The UI should ease correlation between logs and health/performance KPIs as well as alerts.

For organisations having their own Log centralization system (like Splunk or Elasticsearch), MetalK8s should provide some documentation to guide the customer to deploy and configure its own log collection agent.

The following requirements focus on application logs. Audit logs are not part of the requirements.

Requirements

Lightweight

A lightweight tool to store and expose logs is required in order to minimize the HW footprints (CPU, RAM, Disks):

- limited history: Storing the logs for very large period (3 years) is not something metalK8s needs to provide as a feature. This can be achieved using external log centralization system.
- stable ingestion: It is important to guarantee stable ingestion of the logs and less important to guarantee stable performances when browsing/searching the logs. However, peak loads related to complex logs queries should not impact the application workloads and deploying log storage and search service on infra nodes might help achieving this isolation.
- stream indexing: It is not required to have automatic indexing of logs content. Instead, the log centralization service should offer basic features to group/filter logs per tag/metadata defining the log stream.

Accessible from a central UI/API

Platform Admin or Storage Admin can visualize logs from all containers in all namespaces as well as journal logs, including (kubelet, containerd, salt-minion , kernel, initrd, services, etc ...) in Grafana. One can correlate logs and metrics or alerts in one single Grafana Dashboard. Browsing logs can be achieved through a documented API in order to expose logs in MetalK8s UIs or other workloads UIs if needed.

Persistence, Retention

Logs should be stored on a persistent storage. Platform Administrator can configure a max retention period. Some automatic purging mechanism is triggered when logs are older than the retention period or when the persistent storage is about to reach its capacity limit. Purging jobs are logged. A typical and default retention period is of 2 weeks. A formula can be used by solution developers in order to properly size the persistent storage for log centralization (cf documentation requirement).

Horizontally scalable (capacity and performance)

The Platform Administrator can scale the service in order to ingest/query and store more logs. It can be because more workloads are running on the platform or because there is a need to keep bigger history of logs.

Highly Available

Log collection, ingestion, storage and query services can be replicated in order to ensure that we can lose at least one server in the cluster without impacting availability and reliability of the service.

Log Querying

Get all logs for a given period, node(s), pod regex, limited list of predefined labels and free keywords text. Typical Zenko use case: collecting all logs across several components, related to a S3 uniq request. Typical predefined labels are severity and namespace.

Log statistics (nice to have)

The Log centralization service also offers the ability to consume statistics about the logs like the number of occurrences of one type of log during a certain period of time.

Monitorable/Observable service (health, performances and alerts)

The Platform Administrator can monitor capacity usage, ingestion rate, IOPS, latency and bandwidth of the Log centralization service. He can also monitor the health of the service (i.e. if some active alerts exist). He is notified through an alert notification when the service is degraded or unavailable. It can be because the persistent storage is full or unhealthy or because the service does not manage to ingest logs at the requested pace.

Here are few example of situations we would like to detect through those KPIs:

- a workload generating a crazy amount of logs
- a burst of ingested logs
- the log persistent storage getting full
- very slow api responses (impacting usability in Grafana dashboards)
- the ingestion of logs working too slowly

Performances (TBD)

Typical workloads can generate around 1000 logs per second per node.

User Stories

- As a Platform Administrator, I want to browse all MetalK8s containers logs (from all servers) from a unique endpoint, in order to ease distributed K8s and MetalK8s services error investigation.
- As a Platform Administrator, I want to browse non container (kubelet, containerd, salt-minion, cron) logs (from all servers) from a unique endpoint, in order to ease System error investigation.
- As a Storage Administrator, I want to browse all Solution instance containers logs (from all servers) from a unique endpoint, in order to ease Solution instance error investigation.
- As a Platform Administrator, I want to push all containers logs to an external log centralization system, In Order to archive it or aggregate it with other application logs.

(some other US extracted from Loki design doc)

- After receiving an alert on my service and drilling into the query associated with said alert, I want to quickly see the logs associated with the jobs which produced those timeseries at the time of the alert.
- After a pod or node disappears, I want to be able to retrieve logs from just before it died, so I can diagnose why it died.

- After discovering an ongoing issue with my service, I want to extract a metric from some logs and combine it with my existing time series data.
- I have a legacy job which does not expose metrics about errors - it only logs them. I want to build an alert based on the rate of occurrences of errors in the log.

Deployment & Configuration

The log centralization storage service is scheduled on infra nodes. A platform Administrator can operate the service as follows:

- add persistent storage
- configure max retention period
- adjust the number of replicas
- configure the system so that logs are pushed to an external log centralization service
- configure log service alerts (IOPS or ingestion rate, latency, bandwidth, capacity usage) i.e. adjust the thresholds, silence some alerts or configure notifications.

Those operations are accessible from any host able to access the control plane network and are exposed through the centralised cli framework.

When installing or upgrading MetalK8s, the log centralization service is automatically scheduled (as soon as a persistent volume is provisioned) on one infra node.

All configurations of the log centralization service are part of the MetalK8s backup and remains unchanged when performing an upgrade.

During future MetalK8s upgrades, the service stays available (when replicated).

Monitoring

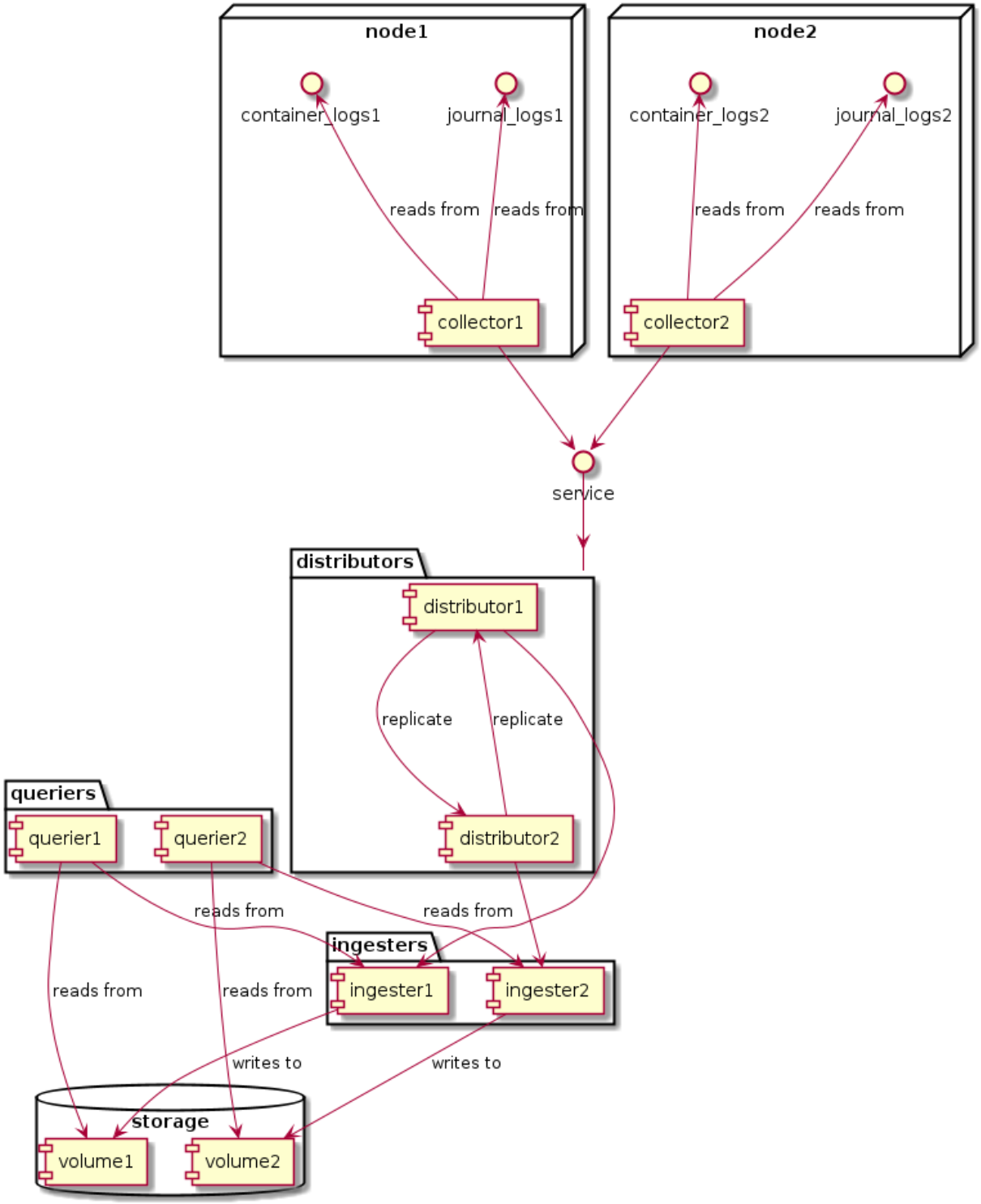
- An alert rule is fired when the log centralization service is not healthy
- The log centralization service is not healthy when log storage is getting full or when service is not able to ingest logs at the right pace.
- IOPS, bandwidth, latency, capacity usage KPIs are available in Prometheus

UI

Logs can be seen in Grafana. Log centralization monitoring information are displayed in the MetalK8s UI overview page. A Grafana dashboard gathering health/performance KPIs, as well as alerts for log centralization service is available when deploying/upgrading MetalK8s.

Components

Our Log Centralization system can be split into several components as follows:



Collector

The collector is responsible for processing logs from all the sources (files, journal, containers, ...), enriching the logs with metadata (labels) coming from parsing/filtering them (e.g. drop record on regexp match), from the context (e.g. file path, host) or by querying external sources such as APIs, then finally forwarding these logs to one or multiple distributors.

Distributor (Router)

The distributor is the component that receives incoming streams from the collectors, it validates them (e.g. labels format, timestamp), then forwards them to the ingester. The distributor can also do parsing/filtering on the streams to enrich them with metadata (labels), route to a specific ingester or even drop them. It can as well do some buffering to avoid nagging the ingester with queries or to wait a bit in case the ingester would be unresponsive for a moment.

Ingester (Storage)

The ingester serves as a buffer between distributor and storage, because writing large chunks of data is more efficient than writing each event individually as it arrives. As such, a querier may need to ask an ingester about what is in the buffer.

Querier

The querier interprets the queries it receives from clients and then asks the ingesters for the corresponding data, then fallback on storage backend if not present in memory and returns it to the clients. It also takes care of deduplication of data because of the replication.

Design Choices

To choose which solution fits best our needs, we did a benchmark of the shortlisted collectors to compare their performances, on a 4 K8s nodes architecture (3 infra + 1 workload), with Loki as backend.

Our choice for the final design has been greatly motivated by these numbers, it represents the global resources consumption for the whole log centralization stack (Loki included).

With 10k events/sec, composed of 10 distinct streams:

	CPU avg in m	**RAM avg in MiB**
promtail	975	928
fluent-bit	790	830
fluent-bit + fluentd	1311	1967

With 10k events/sec, composed of 1000 distinct streams:

	CPU avg in m	**RAM avg in MiB**
promtail	1292	1925
fluent-bit	1040	834
fluent-bit + fluentd	1447	1902

We can see that the Fluent Bit + Loki couple has the smallest impact on resources, but also that the Fluent Bit + Fluentd + Loki architecture seems to offer a better scaling, with the possiblity of keeping less pressure on workload nodes (Fluent Bit), relying more on dedicated infra nodes (Fluentd).

Fluent Bit + Loki

Fluent Bit

We choose Fluent Bit as the collector because it allows to scrape all the logs we need (journal & containers), enriches them with the Kubernetes API and it has a very low resources footprint.

Moreover, it supports multiple backend such as Loki, ES, Splunk, etc. at the same time, which is a very important point if a user also wants to forward the logs to an external log centralization system (e.g. long term archiving).

Check here for the official list of supported outputs.

Loki

Loki has been chosen as the distributor, ingester & querier because, like Fluent Bit, it has a really small impact on resources and is very cost effective regarding storage needs.

It also uses the LogQL syntax for queries, which is pretty close to what we already have with Prometheus and PromQL, so it eases the integration in our tools.

Rejected Design Choices

Promtail + Loki

Promtail has been rejected because, even if it consumes very few resources, it can only integrate with Loki and we need something more versatile, with the ability to interact with different distributors.

Fluentd + Loki

This architecture has been rejected because it means we need 1 Fluentd instance per node, which increases a lot the resources consumption compared to other solutions.

Fluent Bit + Fluentd + Loki

We have considered this architecture as the Fluent Bit + Fluentd couple seems to be a standard in the industry, but we didn't find any reason of keeping Fluentd, apart for its large panel of plugins which we don't really need. Fluent Bit alone seems to be sufficient for what we want to achieve and adding an extra Fluentd means more resources consumption and add unnecessary complexity in the log centralization stack.

Logging Operator

Logging Operator seemed to be a good candidate for the implementation we choose, offering the ability to deploy and configure Fluent Bit and Fluentd, but Fluentd is not optional and seems to have a central place as most of the parsing/filtering is done by this one, which means a bigger footprint on the hardware resources.

Logstash + Elasticsearch

This architecture is probably the most common one, but it has not been taken into consideration because we want to focus on having the minimum resources consumption and these components can really hog RAM & CPU. Beats could be used as the log collector to reduce the impact of Logstash, but Elasticsearch still consumes a lot of resources. Even if this solution offers a lot of powerful functionalities (e.g. distributed storage, full-text indexing), we don't really need them and want to focus on the smallest hardware footprint.

Implementation Details

Deployment

All the components will be deployed using Kubernetes manifests through Salt inside a `metalk8s-logging` Namespace.

Fluent Bit

Fluent Bit will be deployed as a DaemonSet, because we need one collector on each node to be able to collect logs from both the Kubernetes platform, the applications that run on top of it and all the system daemons running alongside (e.g. Salt minion, Kubelet).

This DaemonSet and Fluent Bit configuration will be handled by the Fluent Bit Operator. For this, we need to deploy the manifests found here, using our Salt Kubernetes renderer.

We will then need to also automatically deploy CRs with Fluent Bit default configuration, examples can be found here.

Loki

Loki has 2 deployment mode, either as microservices (with distributor, ingester and querier in distinct pods), either as a monolith. We must use the monolithic mode, because we use filesystem as the storage backend and microservice mode does not support it.

Loki will be deployed as a `StatefulSet` as it needs `PersistentVolume` to write the logs. It will be running only on infra nodes and we need at least 2 replicas of it (except on a single node platform), to ensure its high availability.

As we already did for other components (e.g. Prometheus Operator), the manifests for Loki will be generated statically by rendering the loki helm chart:

```
helm repo add loki https://grafana.github.io/loki/charts
helm repo update
helm fetch -d charts --untar loki/loki
./charts/render.py loki --namespace metalk8s-logging \
  charts/loki.yaml charts/loki/ \
  > salt/metalk8s/addons/loki/deployed/charts.sls
```

Loki Storage

Since we're using filesystem to store Loki's data, we have basically 2 ways for having multiple Loki instances running at the same time, with the same set of data.

Either we do like Prometheus and we store everything on every instance of Loki, but it means we raise the storage, RAM and CPU needs for each additionnal instance, either we use a new experimental feature from Loki 1.5.0 where Loki ingesters use a hash ring and talk between them to route the queries to the right one. This approach needs an external KV store to work such as etcd or Consul.

We choose to use the first approach as the second is not production ready and since we plan to only do short term retention on Loki, the impact on storage will not be that much important. Moreover, it eases the deployment and maintenance since there is not an extra component.

Loki HA

It is possible to set the hash ring store to *memberlist*, so it relies on it and it can discover and communicate with the other Loki instances. This way, it allows any instance to replicate, on some or all the other instances (depending on the replication factor), every logs it receives. Members will be automatically discovered at runtime, by using the Loki headless service as the only member.

Note that, if Loki cannot write enough replicas, it will be stuck (blocking the whole logging stack), waiting to be able to do so. It means, we need to configure the number of replicas depending on how many nodes we want to be able to lose without impacting the logging services and ensure the logging data resiliency. For example, on a 5 nodes cluster, if we configure a replica on each node, we can lose 2 nodes (`(number_of_nodes - 1) / 2`), on a 3 nodes cluster, we can lose 1 node.

This mode is required to be able to handle an instance failure, if we configure fluent-bit to talk to all Loki instances, if one fails, fluent-bit will be stuck, waiting to be able to send its data. This way, as we use the service to talk to one instance which then replicates the data to other instances, if a Loki instance fails, fluent-bit will not know about it and will keep running.

Since we use filesystem as Loki storage backend for the logs, we cannot ensure we will have the data if at some point an instance is down or the instance loses its volume. It means, if we query for this period of time and we hit this very instance we will have no data. Note that, if the instance is only down for few minutes (depending on the quantity of logs of the platform), it will be able to catch up what is still in other ingesters memory and not written yet to the storage backend.

To solve this problem, we either need to be able to share this storage between all the instances (e.g. NFS), but it is probably a bad idea, either we need an extra component in front of the queriers, which takes care of querying each instance, deduplicating the entries and returning them. We have already considered the Loki Query Frontend, but its goal is only to split queries, dispatch the load on the multiple instances and do some caching.

Configuration

Fluent Bit

Fluent Bit needs to be configured to scrape and handle properly journal and containers logs by default.

For containers logs, we want to add the following labels:

- node: the node it comes from
- namespace: the namespace the pod is running in
- instance: the name of the pod

- container: the name of the container

For journal we want these labels:

- node: the node it comes from
- unit: the name of the unit generating these logs

This configuration will also be customizable by the user to be able to add new routes (Output) to push the log streams to.

This configuration will be done through various CRs provided by the Fluent Bit Operator:

- FluentBit: Defines Fluent Bit instances and its associated config
- FluentBitConfig: Select input/filter/output plugins and generates the final config into a Secret
- Input: Defines input config sections
- Filter: Defines filter config sections
- Output: Defines output config sections

For example, if a user wants to forward all Kubernetes logs to an external log centralization system (e.g. Elasticsearch), he will need to define an `Output` CR as follows:

```
kind: Output
metadata:
  name: my-output-to-external-es
  namespace: my-namespace
spec:
  match: kube.*
  es:
    host: 10.0.0.1
    port: 9200
```

More details can be found on Fluent Bit Operator repository and manifest samples are here.

Loki

Loki's configuration is stored as a Secret. we need to expose few parameters to the user for customization (e.g. retention). Since we do not have Operator and CRs for Loki, we will use the CSC mechanism to provide the interface for customization, with a ConfigMap `metalk8s-loki-config` in the `metalk8s-logging` Namespace. CSC is not as powerful as an Operator with CRs (no watch and reconciliation on resources and need to run Salt state manually), but the Loki configuration will not change that much, probably during deployment and to tune few parameters afterwards, so it does not worth to invest on an Operator.

The CSC `ConfigMap` will look like the followings:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: metalk8s-loki-config
  namespace: metalk8s-logging
data:
  config.yaml: |-
    apiVersion: addons.metalk8s.scality.com
    kind: LokiConfig
    spec:
```

(continues on next page)

(continued from previous page)

```
    deployment:
      replicas: 1
    config:
      auth_enabled: false
      chunk_store_config:
        max_look_back_period: 168h
      ingester:
        chunk_block_size: 262144
        chunk_idle_period: 3m
        chunk_retain_period: 1m
        [...]
```

With default values fetched from a YAML file as it is already done for Dex, Alertmanager and Prometheus.

Monitoring

Prometheus

To monitor every services in our log centralization stack, we will need to deploy `ServiceMonitor` object and expose the `/metrics` route of all these components. It will allow Prometheus Operator to configure Prometheus and automatically start scraping these services. For Loki, this can be achieved by adding the following configuration in its helm chart `charts/loki.yaml` values:

```
serviceMonitor:
  enabled: true
  additionalLabels:
    release: "prometheus-operator"
```

For Fluent Bit, we will need to define a `ServiceMonitor` object:

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    app: fluent-bit
    metalk8s.scality.com/monitor: ''
  name: fluent-bit
  namespace: metalk8s-logging
spec:
  endpoints:
  - path: /api/v1/metrics/prometheus
    port: http-metrics
  namespaceSelector:
    matchNames:
    - metalk8s-logging
  selector:
    matchLabels:
      app: fluent-bit
```

We also need to define alert rules based on metrics exposed by these services. This is done deploying new `PrometheusRules` object in `metalk8s-logging` Namespace:

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  labels:
    metalk8s.scality.com/monitor: ''
  name: loki.rules
  namespace: metalk8s-logging
spec:
  groups:
  - name: loki.rules
    rules: <RULES DEFINITION>
```

There is some recording and alert rules defined in the Loki repository that could be used as a base, then we could enrich these rules later when we will have better operational knowledge.

Grafana

To be able to query logs from Loki, we need to add a Grafana datasource, this is done adding a ConfigMap `loki-grafana-datasource` in `metalk8s-monitoring` Namespace as follows:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: loki-grafana-datasource
  namespace: metalk8s-monitoring
  labels:
    grafana_datasource: "1"
data:
  loki-datasource.yaml: |-
    apiVersion: 1
    datasources:
    - name: Loki
      type: loki
      access: proxy
      url: http://loki.loki.svc.cluster.local:3100
      version: 1
```

To display the logs we also need a dashboard, adding a ConfigMap `loki-logs-dashboard` in Namespace `metalk8s-monitoring`:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: loki-logs-dashboard
  namespace: metalk8s-monitoring
  labels:
    grafana_dashboard: "1"
data:
  loki-logs.json: <DASHBOARD DEFINITION>
```

For the dashboard we will use a view with a simple log panel and variables representing labels to filter on. Since journal and kubernetes logs will not have the same labels, we could either have 2 distinct dashboards or 2 log panels in the same. An example of what we want is Loki dashboard.

Note: The `grafana_datasource: "1"` and `grafana_dashboard: "1"` labels are what is used by the Prometheus Operator to retrieve datasource and dashboard for Grafana.

Resources may be deployed in any namespace so long as it contains the above labels.

The *metalk8s.scality.com/grafana-folder-name* annotation on dashboard resources provide control over the folder in which the dashboard is placed in Grafana.

Loki Volume Purge

Even with a max retention period, the logs could grow faster than what was expected and fill up the volume. Since there is no retention based on size in Loki yet, we need to add some specific monitoring (with prediction on volume usage) and alerting to ensure that an administrator will be warned if such a case would happen. The alert message should be clear and provide an URL to a run book to help the administrator resolving the issue.

To fix this issue, the administrator should purge oldest log chunk files from Loki volume, which can be achieved by connecting to the pod and manually removing them. If the growth of logs is not something transient, the administrator should also be advised to lower the retention period or replace the Loki volume by a bigger one.

Iterations

Iteration 1

The goal is to have a working log centralization system, with logs accessible from Grafana:

- Deploy Fluent Bit and Loki
- Customization of Loki with CSC mechanisms
- Document customization of Loki through CSC
- Deploy Grafana datasource & dashboard
- Document the log centralization system (sizing, configuration, ...)
- Simple pre-merge test to ensure the default log pipeline is working

Iteration 2

- Deployment of Fluent Bit with Fluent Bit Operator
- Document customization of Fluent Bit through CRs
- Define Prometheus record and alert rules
- Deploy Loki volumes purge mechanism (TBD)
- Display log centralization system status on the MetalK8s UI
- Post-merge tests to ensure customization is working (replicas, custom parser/filter rules, ...)

Documentation

The sizing section in Introduction page is updated to include log centralization service impact. The sizing rule takes in account the retention period, workloads expected log rate and workload predefined indices. This rule is to be known by solution developers to properly size the service based on the workload properties.

The Post Installation page is updated to indicate that persistent storage is needed for log centralization service.

A new page should be added to explain how to operate the service and how to forward logs to an external log centralization system.

The Cluster Monitoring page is updated to describe the log centralization service.

Test Plan

Log centralization system will be deployed by default with MetalK8s, so its deployment will automatically be tested during pre-merge integration tests. However, we still need to develop specific pytest-bdd test scenario to ensure that the default logging pipeline is fully functionnal, and run it during these pre-merge tests. We will also add more complex tests in post-merge such as configuring specific parsers/filters, scaling the system, etc.

3.1.12 MetalK8s UI adaptation

MetalK8s UI needs adaptation, so that it fits with shell & microfrontends architecture. In the long term, MetalK8s UI always runs with a Shell Nav bar. MetalK8s also provides a secondary lateral Nav bar.

The MetalK8s secondary lateral Nav bar should contains entries that are specific to MetalK8s micro application. Features like login/logout, switching from dark to light mode or switching language are to be exposed in the Shell Nav bar.

The MetalK8s UI exposes MetalK8s operations to users having Platform Administrator role. It also provides High Level Monitoring and Alerting views in order for the Platform Administrator to understand the health and the performances of the platform.

The platform entities to operate and monitor are:

- the Hardware entities (servers, disks, raids, interfaces or Network)
- the OS and k8s system services (kubelet and conatinerd)
- the Kubernetes services (etcd, apiserver, scheduler)
- the Kubernetes entities (Nodes, Volumes)
- the specific MetalK8s services (Monitoring, Alerting, Logging, Authentication, Ingress)

MetalK8s Operations

- Cluster administration (upgrade, downgrade, backup, restore)
- Cluster nodes provisioning (adding/removing nodes, configuring nodes)
- Volume provisioning (adding/removing Volumes, configuring volumes)
- Solution lifecycle (adding/deleting envs, activate/adding solutions)
- MetalK8s services administration (configuring alerts, scaling service)
- roles and security policies management

Some of the operations are not exposed through API and thus cannot be exposed in the UI. The operations that are accessible in the UI are served through either K8S or Salt APIs. Not all operations are critical to have in the UI.

Some operations, such as cluster expansion or volume addition can be time consuming. MetalK8s UI should provide some indication of the progress and ETA. It should also provide easy way to debug and investigate when things go wrong. Thus logs associated to an expansion or volume provisioning should be easily accessible from MetalK8s UI.

Last but not least, all important operations performed on the platform (such as adding/removing a node, upgrading the cluster, ...) should be logged. Browsing the history of those operations should be accessible from MetalK8s UI.

MetalK8s Monitoring

Health

The Health of the system as well as the health of any specific entity of the system, is determined by the existence of an active alert, with a specific severity on it. It is probably needed to filter out some alerts that must not be taken in account to determine the health of any system entity.

The Global health of the platform is either a magic combination of some important entity alerts, either a specific Scality alert delivered with MetalK8s.

Performance

Some perf kpis & charts are available for each entity or group of entities. The aim of those charts is not to replace the Grafana dashboards but to give some high level and synthetic views, focused on what's matter the most for the MetalK8s Platform administrator.

MetalK8s Pages

Overview / Landing Page

As a Platform Administrator, I Want to visualize the key components and services of the platform, In order to understand how the platform works.

As a Platform Administrator, I Want to identify real time and past failures, In order to act on it and fix the it.

As a Platform Administrator, I Want to visualize high level performances of the key HW components, In order to understand real time and past load as well as forecast the load.

When landing on the MetalK8s microapp, an overview of the health and performances of the cluster is displayed. The page is focusing on one single cluster i.e. one single site.

The page is divided in 3 sections:

- global health: the history of alerts and the current health of the system. If the health of the system is degraded, when the Platform Administrator click on it, he is redirected to the alert page which is automatically filtered on the active alerts.
- health of each platform entities: nodes, volumes, k8s master services, monitoring services, logging service, ingress services and auth service. if one entity is not healthy, we can click on it and access the list of active alerts associated with this entity, in the entity page.
- performances: several charts are displayed to show aggregated load, aggregated CPU usage, aggregated RAM usage, aggregated IOPS usage and aggregated throughput for both CP and WP networks. Ideally, Top 10 slowest disks are also displayed.



Nodes Page

As a Platform Administrator, I Want to have a list of the nodes making the cluster as well as detailed information about each node, including node health, in order to identify nodes at risk.

The Nodes Page provides a list of all MetalK8s cluster nodes. For each node, one can see its status and health (most severe active alert), MetalK8s version and roles. It is also from where the Platform Administrator can expand the cluster (i.e. add a node) or access to advanced details about this node.

The list of MetalK8s cluster nodes can be displayed as a table in a first iteration but ideally, they are displayed as cards and the nodes having issues are displayed first. In both cases, a search field is available in order to filter the list of nodes we see in the table or as cards.

When the Platform Administrator click on a node, detailed information of this node is displayed in the right zone of the page (could be bottom zone). That way, the Platform Administrator can focus on one specific node while keeping the table or the cards as a way to navigate to another node.

Node Panel

When clicking on one node in the nodes page, it is possible to access various information about the node through specific tabs.

High level Node information contains its name, its ip, its role, its status. Those High level information are always visible, whatever tab is active.

- health
- volumes
- pods

The health is the tab displayed at first when accessing the node.

health

The list of active and past alerts as well as Key performance indicators over the last 7 days help the Platform Administrator to understand the behaviour of this specific node. Alert table: Name, instance, Severity, Message, Active Since

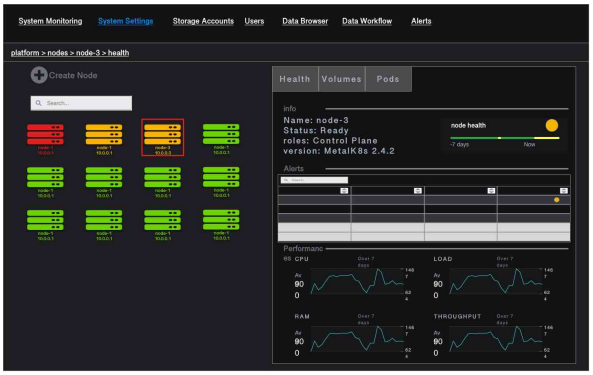
KPIs charts: CPU, Load, Memory, IOPS, WP and CP IO bandwidth. The list of KPI may be different for K8s master nodes and K8s worker nodes.

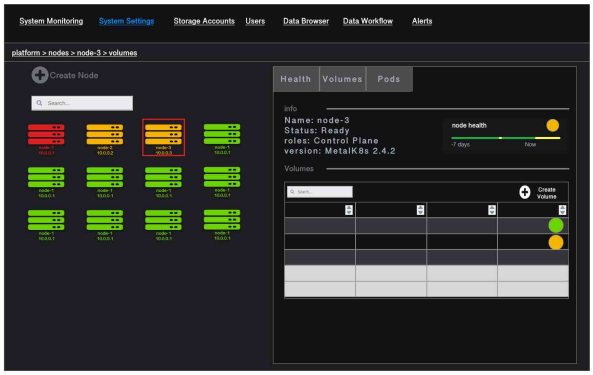
volumes

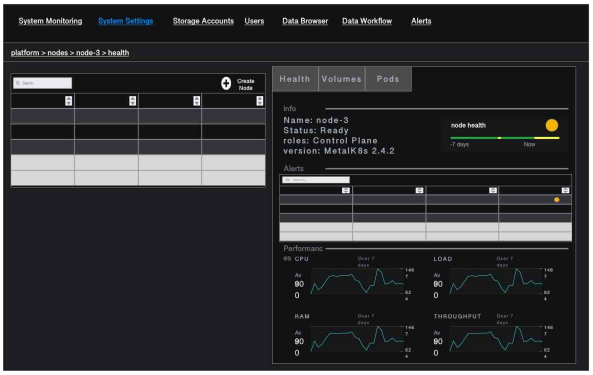
A table with the list of Volumes created on this node. For each volume, status, health, bounded / available, type and size is available. When clicking on one Volume, it is possible to access various information about this specific Volume in the Volume page.

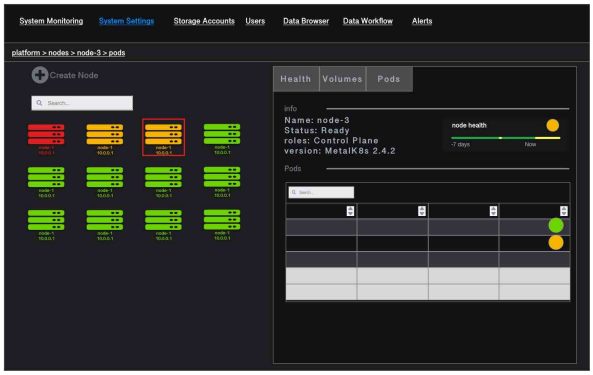
pods

A table with the list of pods scheduled on this node. For each pod, status, health, age, namespace are displayed.









Note: The Node Panel is also where the node creation form would be displayed when the platform administrator clicks on Add Node.

Volumes Page

The Volumes page contains a table with all provisioned Volumes into the system. This view enables to quickly identify the Volumes that are not yet bound to any pod or workload (those Volumes should appear within a dedicated section if they are displayed as cards). It also gives an overview of all created Volumes and their health and status. From the Volumes page it is possible to create a new Volume.

Volumes Table Columns:

- Name
- Node
- Storage class
- Bound: no or pod name if bound

- Status: Available of Failed (if Volume provisioning failed)
- Health: based on active alert existence
- Size (or storage capacity)
- Usage (%utilization) with some gauge bar renderer
- Creation time
- Action (delete/edit) with some icon renderer

Ideally we also want to have avg latency so that we can easily sort Volumes by latency in order to quickly identify slowest disks. If it is too much information to put in the table, we can have another table with only Name, Node, Health and latency information at the bottom of the main volumes table.

Note: One of the duty of the Platform Administrator is also to have a view of the available disks or devices out of which we could create Volumes / PVs. The Platform Administrator will want to know what are the available devices and if they are healthy. Also when a disk is not healthy, the Platform Administrator will want this disk to be easily identified in the data center (i.e. blinking the disk)

Volume Panel

When clicking on one volume in the volumes page, it is possible to access various information about the volume through specific tabs:

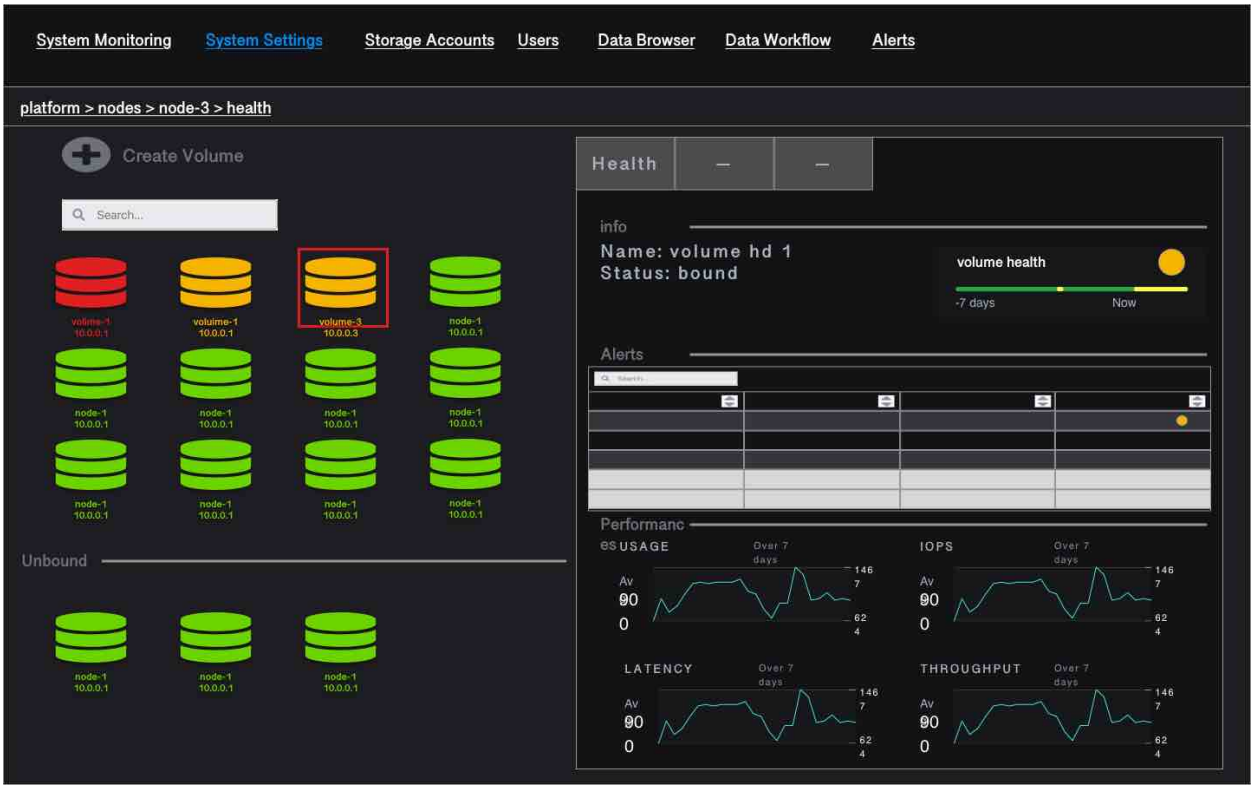
- All table infos
- Labels
- Type (block device or loop device)
- Pod name (if the volume is bound)

High level Volume information contains its name, the node it belongs to, its status, the pod it is bound to. Those High level information are always visible, whatever tab is active.

- health

health

The list of active and past alerts as well as Key performance indicators over the last 7 days help the Platform Administrator to understand the behaviour of this specific node. Alert table: Name, instance, Severity, Message, Active Since KPIs charts: Usage (used, total, available), IOPS, IO Latency and IO bandwidth.



Note: The Volume Panel is also where the volume creation form would be displayed when the platform administrator clicks on Add Volume.

Cards vs Table for Nodes ad Volumes

Cards offer a more sexy way of presenting instances. We can even layout or group cards according to some business logic criteria. However we can't display lot of information or sort the cards. The table is may be less fancy, however it helps the user to visualize lot of information at the same time, it usually embeds out of the box sorting, filtering and top 10 capabilities. Also each cell can be rendered using fancy components (an not only text).

MetalK8s services

As a reminder, the list of MetalK8s services are the following:

- k8s master
- bootstrap (OCI registry & salt)
- monitoring
- logging
- ingress
- auth

As a Platform Administrator, I want to make sure all MetalK8s services are running properly, in order to make sure Solution instances can run properly and other admin users can perform their tasks.

As a Platform Administrator I want to understand on which nodes, each service sub components are scheduled and what are the Volumes involved if any, In order to know HW entities that may have an impact on it.

As an example, monitoring service is made of Prometheus to store all statistics as well as Alert Manager to manage the alerts.

As a Platform Administrator I want to know if there are some actives alerts on a service and I want to visualize the history of alerts In order to act on it to fix the issue.

As a Platform Administrator I want to to know how a service is behaving in terms of performances (CPU, Load, Memory, IO), In order to anticipate potential failure events.

As a Platform Administrator, I want to scale up/down one service, In order to better handle the load.

The Platform Administrator may also need dedicated pages in order to configure the various services (mainly thinking about alerting, auth and ingress)

Environment & solution Page

As a Platform administrator I want to create an MK8s environment (production / Staging / Test…) in order to install the Scality data and storage management solutions

As a Storage administrator I want to add and manage solution lifecycle within an environment in order to upgrade/downgrade the Scality data and storage management solutions components.

In the long term, those functionalities may need to be exposed outside of metalK8s, especially if we need to deploy environment and solutions across multiple cluster or sites.

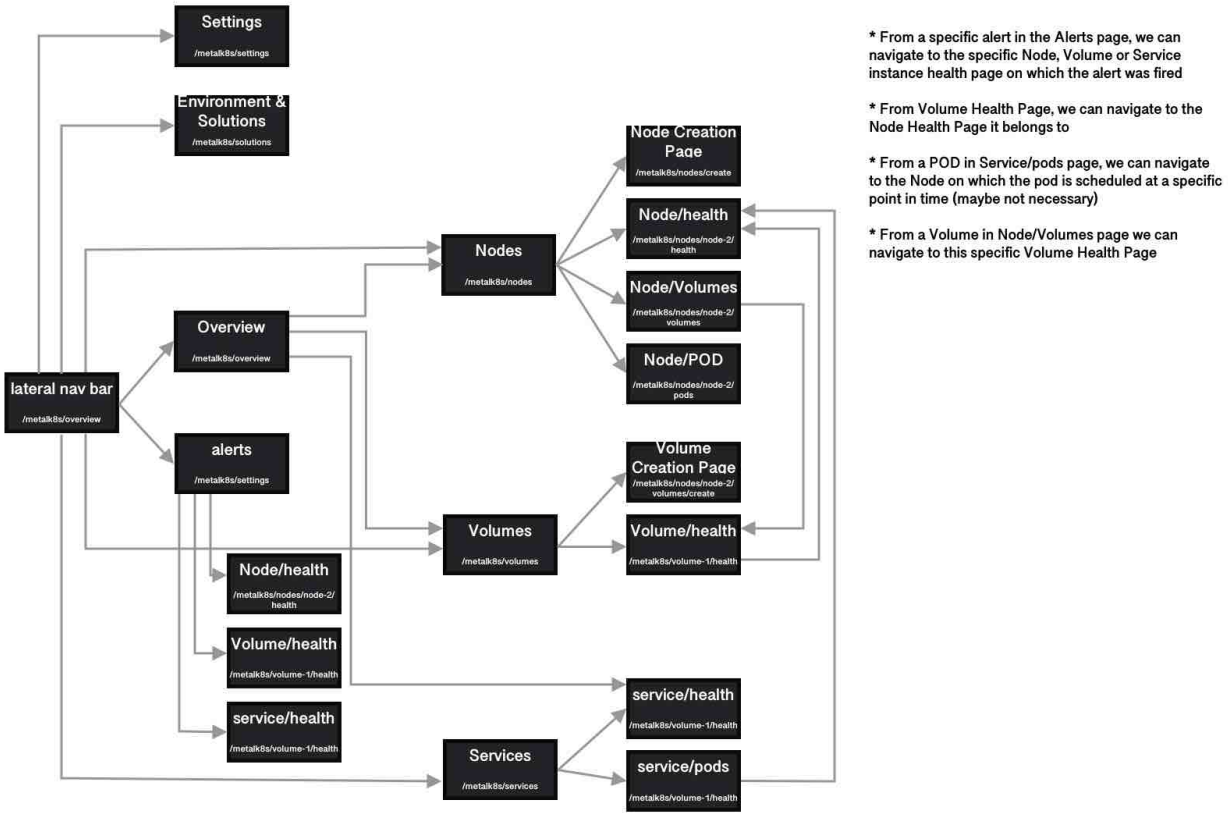
System Settings Page

No detailed functionalities for now. This could be from where the Platform Administrator would trigger platform upgrade, downgrade, restore or backup.

Alerts Page

From this page the Platform Administrator can visualize all past and current alerts belonging to any entity of the platform. When clicking on one specific alert, the user is redirected the specific entity / health page on which the alert was fired.

Overall Navigation



3.1.13 Monitoring

Context

In a MetalK8s cluster, almost each component (including applications running on top of it) generates metrics that allow to monitor its health or expose internal information.

These metrics can be scrapped and then displayed to the end user in a fashion way to give him an easily readable overview of the cluster and applications health and status.

Moreover, the metrics can also be used to send alerts depending on their values, allowing to quickly see if there is any issue or strange behavior on a component.

Here, we want to describe the monitoring stack deployed in MetalK8s.

Goals

- Highly available metric history
- Trigger alerts based on metric values
- Visualization of metrics through charts
- Easily scrape new metrics (from new MetalK8s pods or from workload pods running on top of MetalK8s)
- Configuration of alert routing and some existing alerts
- Possibility to add extra alerts

Design Design Choices

In order to scrape metrics, trigger alerts and visualize metrics history we decided to use Prometheus, AlertManager and Grafana since it is the de facto standard for Kubernetes monitoring and is well integrated with almost any application running on top of Kubernetes.

We use kube-prometheus which embeds the prometheus-operator that deploys the various components but also allows to easily monitor new metrics, add new Grafana dashboards and datasources.

But Prometheus does not work as a cluster which means that to have proper HA we need to duplicate information on several Prometheus instances.

Either each Prometheus instance monitors only a part of all nodes (with some overlaps in order to have HA), but then to visualize information we need to reach the right Prometheus instance depending on the node we want to see.

Either each Prometheus instance monitors all nodes, but even in this case if for whatever reason one Prometheus is down for a short time we will have a "blank" in the resulting graphs while the information is available in the other Prometheus instances.

To solve this we decided to use Thanos, so that we have a Thanos side-car on every Prometheus instances and a global Thanos querier that will query all the Prometheus instances in order to get metrics asked. And then it will deduplicate series, that come from different Prometheus instances, by merging all of them in a single time series. Check the Thanos querier documentation if you want more information about how it works.

Thanos queries are exactly the same as Prometheus ones they both use PromQL.

For the moment we only consider Thanos as a querier and we will not configure any long term retention, but it's a Thanos capability that can be interesting in the future.

Rejected Choices

Federated Prometheus

Prometheus can federate other Prometheus instances, it basically fetches all metrics from other Prometheus instances instead of getting them directly from Pod metric endpoints.

It allows to have all metrics available in a single point, with a small network impact.

This approach was rejected because it means we need to store all metrics in yet another Prometheus instance and it does not properly scale if we really want all metrics in this federated Prometheus instance.

Cortex

Cortex allows to have highly available metrics history, all Prometheus instances just push all metrics to Cortex.

Cortex works almost the same way as a Federated Prometheus except that it runs as a cluster, so it is more scalable since you can have several Cortex replicas that only own a part of the metrics. The Cortex Querier will then retrieve the metrics from the right Cortex instance.

This approach was rejected because it needs a dedicated DB to store the Cortex hash and it consumes more storage, it is a bit excessive for a single Kubernetes cluster (which is the intend of MetalK8s) and Cortex cannot be used as a querier only.

Implementations Details

In order to deploy Thanos Prometheus side-car and deploy the Thanos querier that interact with those side-cars we need to add new Salt states in MetalK8s.

Those salt states come from Kubernetes Prometheus stack helm chart and Thanos helm chart as described in the Thanos readme, then those helm charts are rendered to salt states using the `render.py` script from `charts` directory.

Thanos will only be used as a querier for the moment so we do not need any specific configuration for it.

Create a Grafana datasource for Thanos querier and use it for all dashboards that need to query Prometheus metrics.

Expose Thanos querier through the *metalk8s-ui-proxies-http* Ingress object to be able to query Prometheus metrics from the MetalK8s UI.

Documentation

In the Installation Guide:

- Document how to access Grafana

In the Operational Guide:

- Document how this monitoring stack works
- Document what are the default alerts deployed by MetalK8s
- Document how to perform a Prometheus snapshot
- Document how to configure Prometheus through CSC

Test Plan

Add test scenarios for monitoring stack using pytest-bdd framework to ensure the correct behavior of this feature.

- Ensure that every deployed Pods are properly running after installation
- Ensure that metrics are properly scrapped on every Prometheus instances
- Ensure that alerts are properly raised
- Ensure that we can query metrics from Thanos Querier

3.1.14 Requirements

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.

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.

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.

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

Storage

TODO

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).

3.1.15 Solutions

Context

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

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

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

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

Requirements

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

User Stories

Application import

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

Application deployment and lifecycle

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

Multiple instances of an application

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

Application development

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

Application packaging

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

Application validation

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

Design Choices

Solutions

What's a Solution

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

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

Solution Configuration

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

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

Here is how it will look:

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

In this configuration file, no explicit information about the contents of archives should appear. When read by Salt at import time, the archive metadata will be discovered from the archive itself using a `manifest.yaml` file at the root of the archive, with the following format:

```
apiVersion: solutions.metalk8s.scality.com/v1alpha1
kind: Solution
metadata:
  annotations:
    solutions.metalk8s.scality.com/display-name: Solution Name
  labels: {}
  name: solution-name
spec:
  images:
    - some-extra-image:2.0.0
    - solution-name-operator:1.0.0
    - solution-name-ui:1.0.0
  operator:
    image:
      name: solution-name-operator
      tag: 1.0.0
  version: 1.0.0
```

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

The external pillar will be structured as follows:

```
metalk8s:
  solutions:
    available:
      solution-name:
        - active: True
          archive: /path/to/solution/archive.iso
          manifest:
            # The content of Solution manifest.yaml
            apiVersion: solutions.metalk8s.scality.com/v1alpha1
            kind: Solution
            metadata:
              annotations:
                solutions.metalk8s.scality.com/display-name: Solution Name
              labels: {}
              name: solution-name
            spec:
              images:
                - some-extra-image:2.0.0
                - solution-name-operator:1.0.0
                - solution-name-ui:1.0.0
```

(continues on next page)

(continued from previous page)

```
            operator:
              image:
                name: solution-name-operator
                tag: 1.0.0
            version: 1.0.0
        id: solution-name-1.0.0
        mountpoint: /srv/scality/solution-name-1.0.0
        name: Solution Name
        version: 1.0.0
  config:
    # Content of /etc/metalk8s/solutions.yaml (SolutionsConfiguration)
    apiVersion: solutions.metalk8s.scality.com/v1alpha1
    kind: SolutionsConfiguration
    archives:
      - /path/to/solutions/archive.iso
    active:
      solution-name: X.Y.Z-suffix (or 'latest')
  environments:
    # Fetched from namespaces with label
    # solutions.metalk8s.scality.com/environment
    env-name:
      # Fetched from namespace annotations
      # solutions.metalk8s.scality.com/environment-description
      description: Environment description
      namespaces:
        solution-a-namespace:
          # Data of metalk8s-environment ConfigMap from this namespace
          config:
            solution-name: 1.0.0
        solution-b-namespace:
          config: {}
```

Archive format

The archive will be packaged as an ISO image.

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

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

Solution archive will be structured as follows:

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

(continues on next page)

(continued from previous page)

```
├── operator
|   └── deploy
|       ├── crds
|       |   └── some_crd_name.yaml
|       └── role.yaml
└── registry-config.inc
```

OCI Images registry

Every container images from Solution archive will be exposed as a single repository through MetalK8s registry. The name of this repository will be computed from the Solution manifest `<metadata.name>-<spec.version>`.

Operator Configuration

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

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

Solution environment

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

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

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

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

This *ConfigMap* will be structured as follows:

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

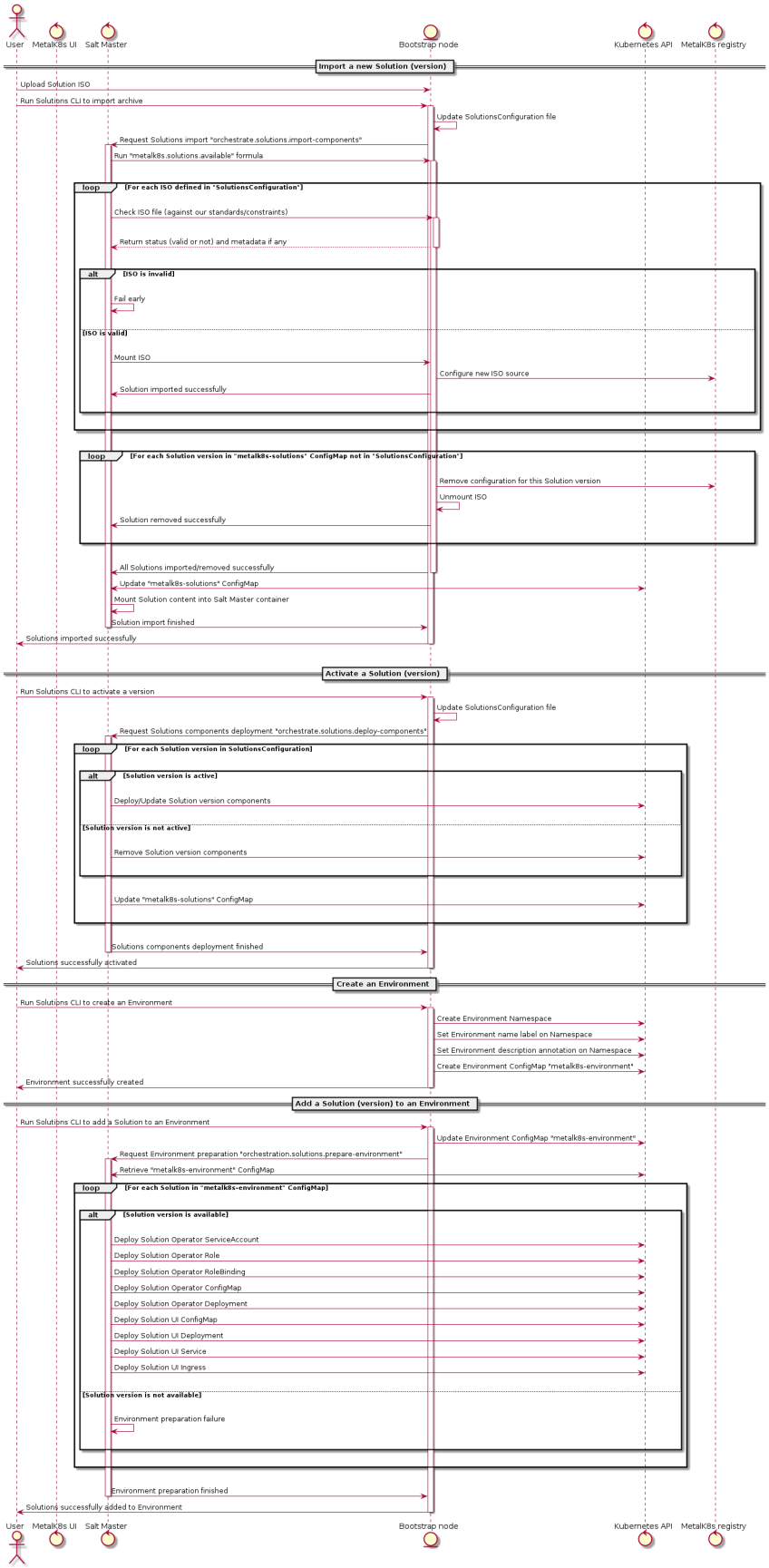
`Environments` will be created by a CLI tool or through the MetalK8s Environment page (both should be available), prior to deploy Solutions.

Solution management

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

Interaction diagram

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


User
MetalK8s UI
Salt Master
Bootstrap node
Kubernetes API
MetalK8s registry
Import a new Solution (version)
Upload Solution ISO
Run Solutions CLI to import archive
Update SolutionsConfiguration file
Request Solutions import "orchestrate.solutions.import-components"
Run "metalk8s.solutions.available" formula
loop
[For each ISO defined in "SolutionsConfiguration"]
Check ISO file (against our standards/constraints)
Return status (valid or not) and metadata if any
alt
[ISO is invalid]
Fail early
[ISO is valid]
Mount ISO
Configure new ISO source
Solution imported successfully
loop
[For each Solution version in "metalk8s-solutions" ConfigMap not in "SolutionsConfiguration"]
Remove configuration for this Solution version
Unmount ISO
Solution removed successfully
All Solutions imported/removed successfully
Update "metalk8s-solutions" ConfigMap
Mount Solution content into Salt Master container
Solution import finished
Solutions imported successfully
Activate a Solution (version)
Run Solutions CLI to activate a version
Update SolutionsConfiguration file
Request Solutions components deployment "orchestrate.solutions.deploy-components"
loop
[For each Solution version in SolutionsConfiguration]
alt
[Solution version is active]
Deploy/Update Solution version components
[Solution version is not active]
Remove Solution version components
Update "metalk8s-solutions" ConfigMap
Solutions components deployment finished
Solutions successfully activated
Create an Environment
Run Solutions CLI to create an Environment
Create Environment Namespace
Set Environment name label on Namespace
Set Environment description annotation on Namespace
Create Environment ConfigMap "metalk8s-environment"
Environment successfully created
Add a Solution (version) to an Environment
Run Solutions CLI to add a Solution to an Environment
Update Environment ConfigMap "metalk8s-environment"
Request Environment preparation "orchestration.solutions.prepare-environment"
Retrieve "metalk8s-environment" ConfigMap
loop
[For each Solution in "metalk8s-environment" ConfigMap]
alt
[Solution version is available]
Deploy Solution Operator ServiceAccount
Deploy Solution Operator Role
Deploy Solution Operator RoleBinding
Deploy Solution Operator ConfigMap
Deploy Solution Operator Deployment
Deploy Solution UI ConfigMap
Deploy Solution UI Deployment
Deploy Solution UI Service
Deploy Solution UI Ingress
[Solution version is not available]
Environment preparation failure
Environment preparation finished
Solutions successfully added to Environment

Rejected design choices

CNAB

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

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

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

Implementation Details

Iteration 1

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

Iteration 2

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

Documentation

In the Operational Guide:

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

In the Developer Guide:

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

Test Plan

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

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

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

3.1.16 Volume Management

Abstract

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

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

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

Scope

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

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.

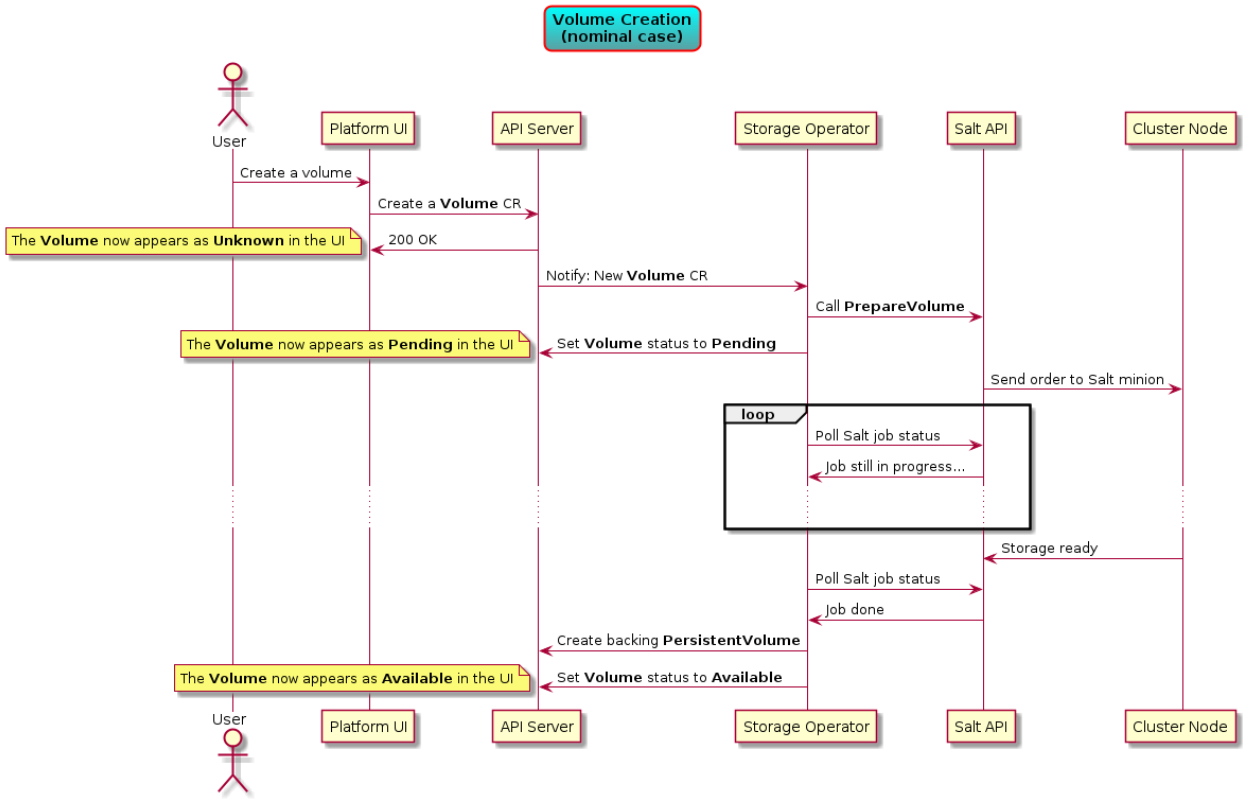
Component Interactions

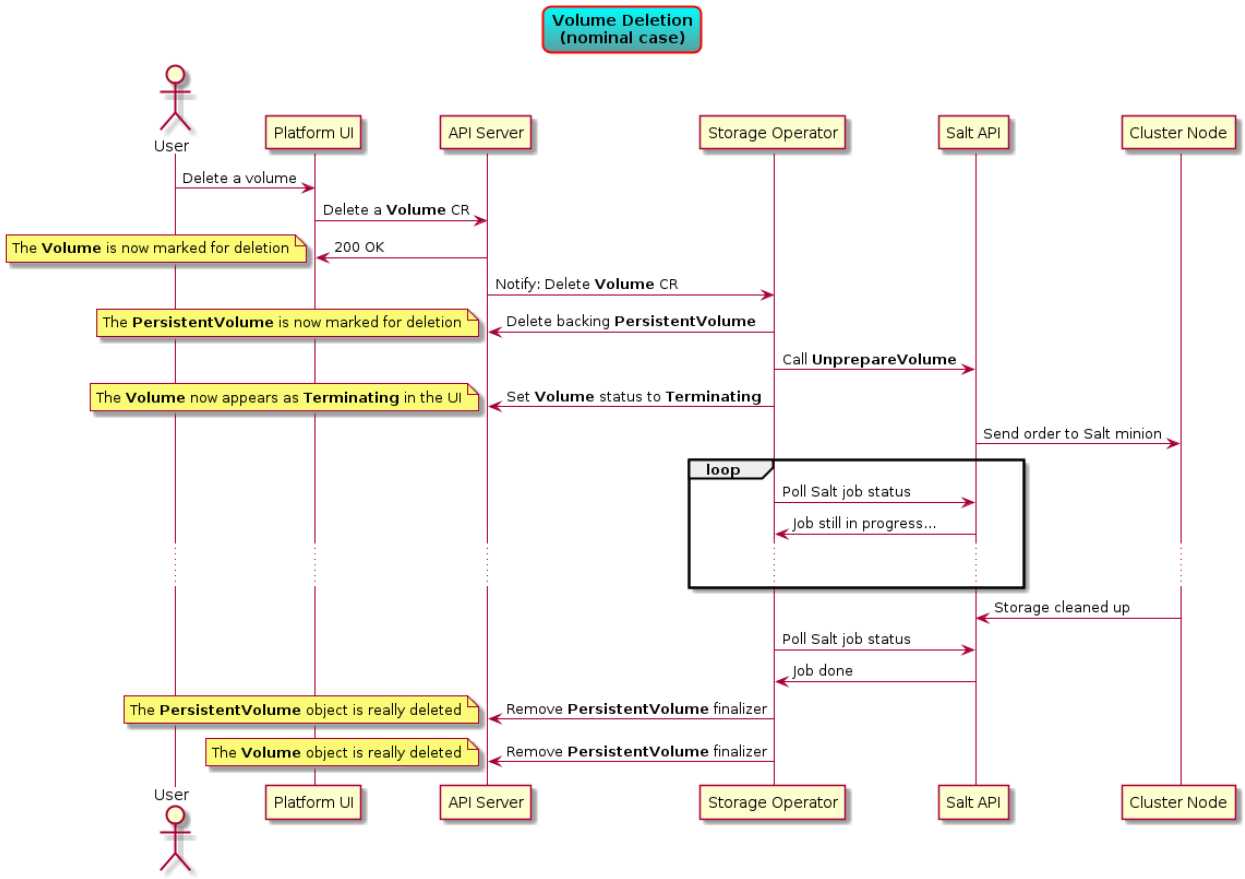
User will create Metalk8s volumes through the Platform UI.

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**.





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).

Persistent block device naming

In order to have a reliable automount through kubelet, we need to create the underlying **PersistentVolume** using a persistent name for the backing storage device. We use different strategies according to the **Volume** type:

- **sparseLoopDevice** and **rawBlockDevice** with a filesystem: during the formatting, we set the filesystem UUID to the **Volume** UUID and use `dev/disk/by-uuid/<volume-uuid>` as device path.
- **sparseLoopDevice** without filesystem: we create a GUID Partition Table on the sparse file and create a single partition encompassing the whole device, setting the GUID of the partition to the **Volume** UUID. We can then use `/dev/disk/by-partuuid/<volume-uuid>` as device path.
- **rawBlockDevice** without filesystem:
  - the **rawBlockDevice** is a disk (e.g. `/dev/sdb`): we use the same strategy as above.
  - the **rawBlockDevice** is a partition (e.g. `/dev/sdb1`): we change the partition GUID using the **Volume** UUID and use `/dev/disk/by-partuuid/<volume-uuid>` as device path.
  - The **rawBlockDevice** is a LVM volume: we use the existing LVM UUID and use `/dev/disk/by-id/dm-uuid-LVM-<lvm-uuid>` as device path.

Operator Reconciliation Loop

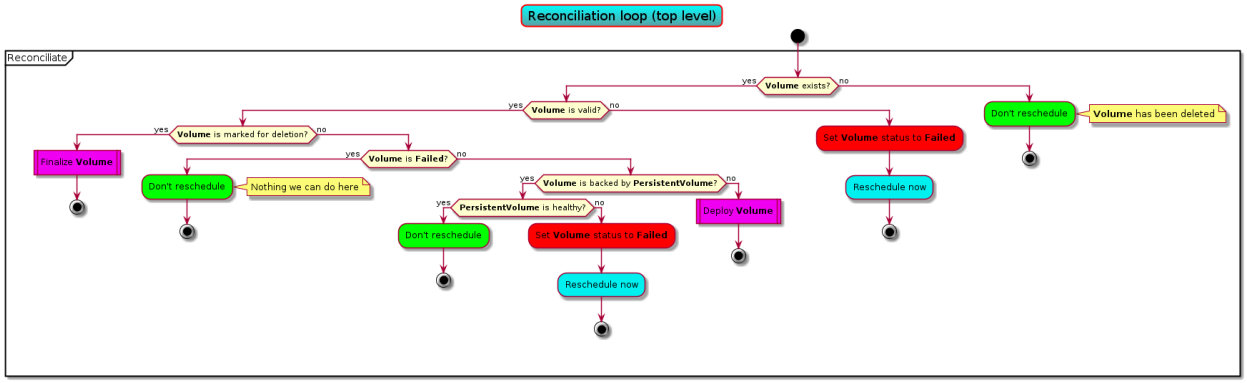
Reconciliation Loop (Top Level)

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**.

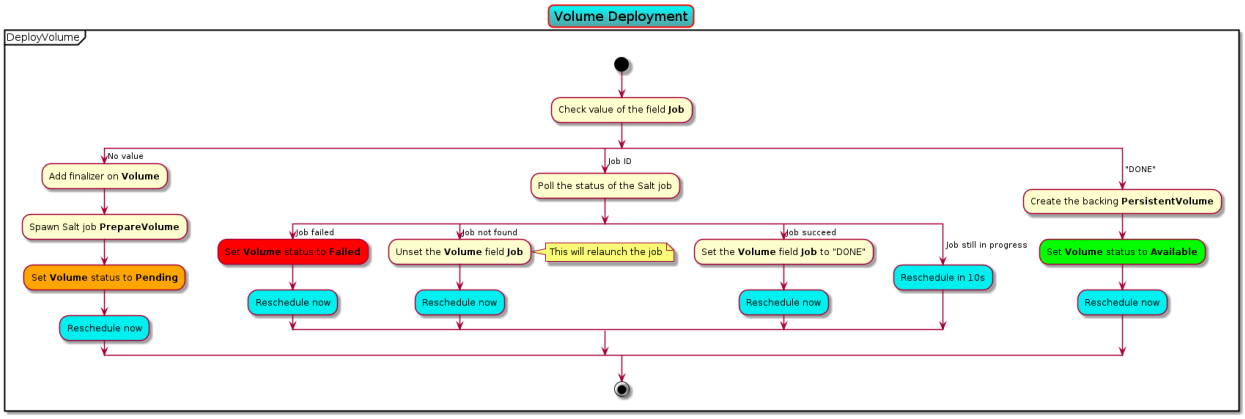
If the **Volume** object has no value in its `Job` field, it means that the deployment hasn't started, thus the operator will set a finalizer on the **Volume** object and then start the preparation of the storage using an asynchronous Salt call (which gives a job ID) before rescheduling the request to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

If we do have a job ID, then something is in progress and we monitor it until it's over. If it has ended with an error, we move the volume into a failed state.

Otherwise we make another asynchronous Salt call to get information (size, persistent path, ...) on the backing storage device (the polling is done exactly as described above).

If we successfully retrieved the storage device information, we proceed with the **PersistentVolume** creation, taking care of putting a finalizer on the **PersistentVolume** (so that its lifetime is tied to ours) and setting ourself as the owner of the **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).



Steady state

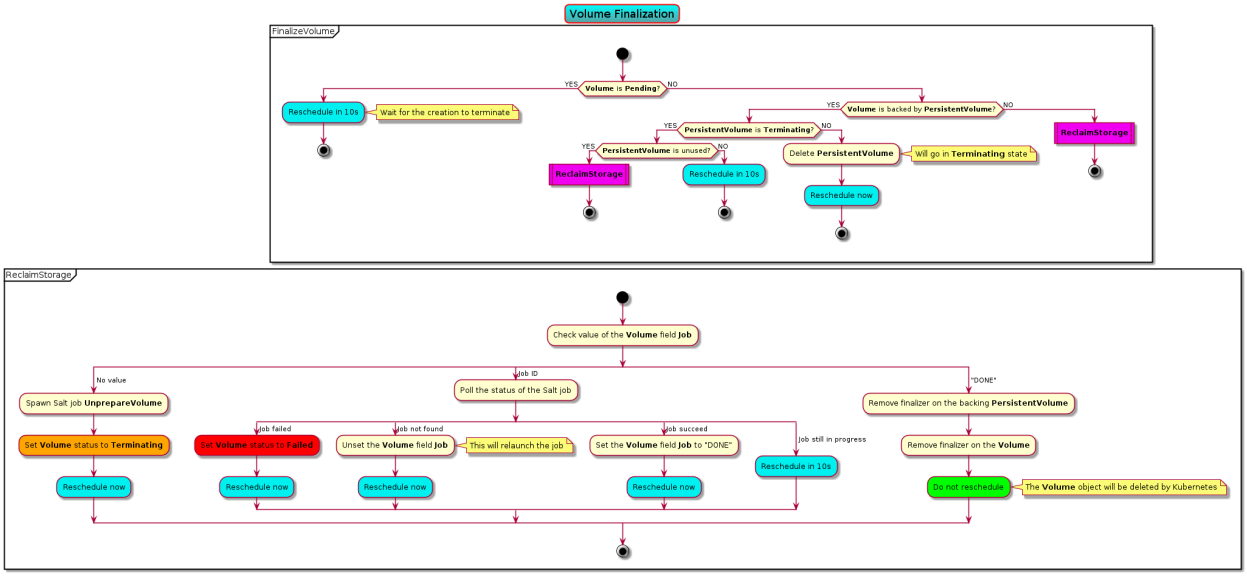
Once the volume is deployed, we update, with a synchronous Salt call, the *deviceName* status field at each reconciliation loop iteration. This field contains the name of the underlying block device (as found under */dev*).

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

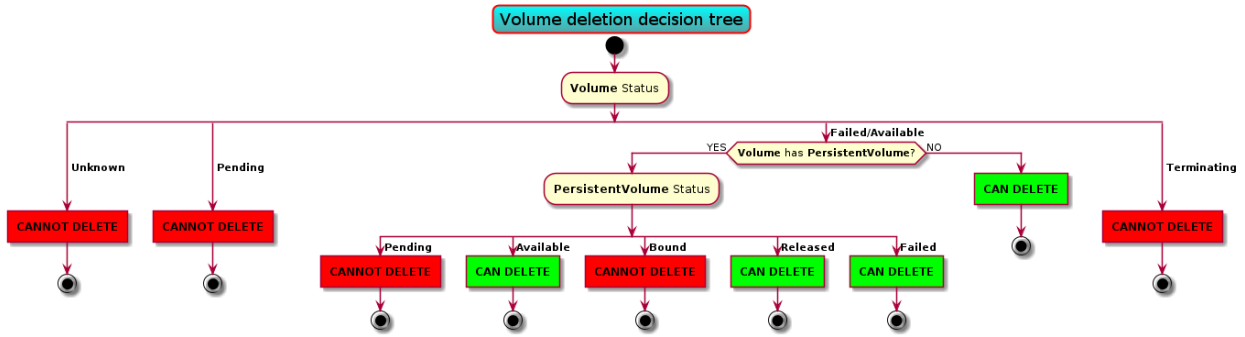
A volume should be deletable from the UI when it's deletable from a user point of view (you can always delete an object from the API), i.e. when deleting the object will trigger an "immediate" deletion (i.e. the object won't be retained).

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**)



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

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.

3.2 How to build MetalK8s

3.2.1 Requirements

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

Mandatory

- Python 3.6 or higher: our buildchain is Python-based
- docker 17.03 or higher: to build some images locally
- skopeo, 0.1.19 or higher: to save local and remote images
- hardlink: to de-duplicate images layers
- mkisofs: to create the MetalK8s ISO
- *implantisomd5* from the isomd5sum package: to embed an MD5 checksum in the generated ISO, allowing for its integrity to be checked

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

Development

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

3.2.2 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`

The execution of tasks is printed to standard output. Each line shows a state prefix, the task type, the task name, and an optional duration.

Tasks can be in one of four states:

- `STARTED`: the execution started
- `SUCCESS`: the execution suceeded
- `ERROR`: the execution failed (more details will be printed to standard error)
- `SKIPPED`: the task is skipped because already up-to-date

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

3.2.3 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
```

3.2.4 Buildchain features

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

doit tabcompletion

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

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`.

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`.

doit info

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

Example:

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

_build_rpm_packages:calico-cni-plugin/srpm

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

status     : up-to-date

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

(continues on next page)

(continued from previous page)

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

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

Wildcard selection

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

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

3.3 How to run components locally

3.3.1 Running a cluster locally

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

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
- provision sparse-file Volumes for Prometheus and Alertmanager to run on this bootstrap node

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

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

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

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

3.3.2 Running the storage operator locally

Requirements

- Go (1.13 or higher) and operator-sdk (0.17 or higher): to build the Kubernetes Operators
- Mercurial: some Go dependencies are downloaded from Mercurial repositories.

Prerequisites

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

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
```

3.3.3 Running the platform UI locally

Requirements

- Node.js, 14.16

Prerequisites

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

Procedure

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

```
kubectl --kubeconfig /etc/kubernetes/admin.conf \
    edit cm -n metalk8s-auth metalk8s-dex-config
```

This will allow you to register localhost:8084 as a valid authentication target. To do so add the following sections under config.yaml:

```
web:
  allowedOrigins: ["*"]
staticClients:
  - id: metalk8s-ui
    name: MetalK8s UI
    redirectURIs:
      - https://<bootstrap_control_plane_ip>:8443/
      - http://localhost:8084/
    secret: ybrMJpVMQxsiZw26MhJzCjA2ut
```

You can retrieve the `bootstrap_control_plane_ip` by running:

```
salt-call grains.get metalk8s:control_plane_ip
```

1. Apply the changes using Salt:

```
VERSION="your version (e.g. 2.9.1-dev)"
SALT_MASTER=$(kubectl \
    --kubeconfig /etc/kubernetes/admin.conf get pods \
    -n kube-system -l app=salt-master \
    -o jsonpath='{.items[0].metadata.name}')
kubectl --kubeconfig /etc/kubernetes/admin.conf exec \
    "$SALT_MASTER" -c salt-master -n kube-system -- \
    salt-run state.sls metalk8s.addons.dex.deployed saltenv=metalk8s-$VERSION
```

1. Enable CORS requests:

```
kubectl --kubeconfig /etc/kubernetes/admin.conf patch ingress \
    -n metalk8s-ui \
    metalk8s-ui-proxies-https \
    --patch '{
        "metadata": {
            "annotations": {
                "nginx.ingress.kubernetes.io/enable-cors": "true",
                "nginx.ingress.kubernetes.io/cors-allow-headers":
                "DNT,X-CustomHeader,Keep-Alive,User-Agent,X-Requested-With,If-Modified-
↪Since,Cache-Control,Content-Type,Authorization,x-auth-token"
            }
        }
    }'

kubectl --kubeconfig /etc/kubernetes/admin.conf patch ingress \
    -n metalk8s-ui \
    metalk8s-ui-proxies-http \
```

(continues on next page)

(continued from previous page)

```
    --patch '{
        "metadata": {
            "annotations": {
                "nginx.ingress.kubernetes.io/enable-cors": "true",
                "nginx.ingress.kubernetes.io/cors-allow-headers":
                "DNT,X-CustomHeader,Keep-Alive,User-Agent,X-Requested-With,If-Modified-
↪Since,Cache-Control,Content-Type,Authorization,x-auth-token"
            }
        }
    }'
```

1. In *webpack.dev.js* edit the value of *controlPlaneIP* and provide your cluster bootstrap node's control plane IP
2. Run ShellUI with `cd shell-ui; npm run start`
3. Run the UI with `cd ui; npm run start:ui`
4. Access the UI by browsing to `http://localhost:8084`

3.4 Deploy new MetalK8s image

3.4.1 Upgrade a Platform with the latest dev build

Prerequisites

- A Platform already installed with all pod up & running
- A new metalk8s.iso image with a higher version

Procedure

If upgrading from a lower patch, minor or major version just follow the *standard upgrade procedure*

If upgrading from the same patch, minor and major version:

1. Upload the new metalk8s.iso on the bootstrap node
2. Locate how metalk8s.iso is mounted

   ```
   grep metalk8s-2.11.7 /etc/fstab
   /root/metalk8s.iso          /srv/scality/metalk8s-2.11.7 iso9660 nofail,ro       0 0
   ```

3. Unmount the current metalk8s.iso

   ```
   umount /srv/scality/metalk8s-2.11.7
   ```

4. Copy the new metalk8s.iso in place of the old one

   ```
   cp metalk8s.iso /root/metalk8s.iso
   ```

5. Mount the new metalk8s.iso

   ```
   mount /root/metalk8s.iso /srv/scality/metalk8s-2.11.7
   ```

6. Stop salt-master container on bootstrap node

```
crictl stop $(crictl ps -q --label io.kubernetes.container.name=salt-master --
↪state Running)
```

7. Provision the new metalk8s ISO content

   `/srv/scality/metalk8s-2.11.7/iso-manager.sh`

8. Upgrade the cluster

   `/srv/scality/metalk8s-2.11.7/upgrade.sh`

3.5 Development

3.5.1 Developing Tests

Continuous Testing

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
```

(continues on next page)

(continued from previous page)

```
    # ...
    # 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
```

Salt Formulas Unit Testing

Introduction

This test suite aims to provide full coverage of Salt formulas rendering, intending to protect formula designers from avoidable mistakes.

The tests are built using:

- `pytest` fixtures to simulate a real rendering context
- `jinja2` as a library, to extract the Jinja AST and infer coverage from rendering passes (more renderers may be covered in the future, if the need arises)

Goals / Non-goals

The tests written here are designed as **unit tests**, covering only the rendering functionality.

As such, a test will ensure that the *render-time* behaviour is tested (and all branches are covered), and will verify the result's validity.

Providing coverage information will be paramount, to build the desired protection (enforcing coverage will also improve the confidence in newly created formulas).

These tests do not include integration or end-to-end evaluation of the formulas. Checking the validity of the rendered contents is only a possibility, and not a primary goal for this test suite.

Implementation Plan

Generic Test Behaviour

Every rendering test will behave the same way:

1. Set up test fixtures
2. Read a formula
3. Render it from the desired context
4. Run some validity check(s) on the result

Configuration

Not all formulas are made the same, and some will only be renderable in specific contexts.

To handle this situation, we define a series of supported context options to customize the test fixtures, and a configuration file for describing the context(s) supported by each formula.

Since we do not want to specify these for each and every formula, we define a configuration structure, based on YAML, which builds on the natural hierarchy of our Salt formulas.

Each level in the hierarchy can define test cases, with the special `_cases` key. This key contains a map, where keys are (partial) test case identifiers, and values are describing the context for each test case. The root-level `default_case` defines the default test case applied to all tests, unless specified otherwise. The `default_case` also serves as default configuration of any test case from which overrides are applied.

Assuming we have some options for specifying the target minion's OS and its configured Volumes, here is how we could use these in a configuration file:

```
default_case:
  os: CentOS/7
  volumes: none

map.jinja:
  # All cases defined here will use `volumes = "none"`
  _cases:
    "CentOS 7":
      os: CentOS/7
    "RHEL 7":
      os: RHEL/7
    "RHEL 8":
      os: RHEL/8

volumes:
  prepared.sls:
    # All cases defined here will use `os = "CentOS/7"`
    _cases:
      "No Volume to prepare":
        volumes: none
      "Only sparse Volumes to prepare":
        volumes: sparse
      "Only block Volumes to prepare":
        volumes: block
      "Mix of sparse and block Volumes to prepare":
        volumes: mix
```

To further generate interesting test cases, each entry in the `_cases` map supports a `_subcases` key, which then behaves as a basic `_cases` map. Assuming we have options for choosing a deployment architecture and for passing overrides to the available pillar, here is how it could look like:

```
_cases:
  "Single node":
    architecture: single-node
    _subcases:
      "Bootstrapping (no nodes in pillar)":
        pillar_overrides:
          metalk8s: { nodes: {} }
      "Version mismatch":
        pillar_overrides:
          metalk8s:
            nodes:
              bootstrap:
                version: 2.6.0

  "Multi nodes":
    architecture: multi-nodes
    _subcases:
      # No additional option
      "All minions match": {}
      "Some minion versions mismatch":
```

(continues on next page)

(continued from previous page)

```
      pillar_overrides:
        metalk8s:
          nodes:
            master-1:
              version: 2.6.0
    "All minion versions mismatch":
      pillar_overrides:
        metalk8s:
          nodes:
            bootstrap:
              version: 2.6.0
            master-1:
              version: 2.6.0
            master-2:
              version: 2.6.0
```

The full configuration currently used is *included below for reference*.

Fixtures

Formulas require some context to be available to render. This context includes:

- Static information, like `grains` or `pillar` data
- Dynamic methods, through `salt` execution modules
- Extended Jinja functionality, through custom filters (likely provided by Salt)

The `pytest` fixtures defined with the tests should allow to setup a rendering context through composition. Dynamic `salt` functions should attempt to derive their results from other static fixtures when possible.

Validity

Important: This is not yet implemented.

The result of a formula rendering shall be validated by the tests. As most formulas use the `jinja|yaml` rendering pipeline, the first validity check implemented will only attempt to load the result as a YAML data structure.

Later improvements may add:

- Structure validation (result is a map of string keys to list values, where each list contains either strings or single-key maps)
- Resolution of `include` statements
- Validity of requisite IDs (`require`, `onchanges`, etc.)

Coverage

Important: This is not yet implemented.

Obtaining coverage information for non-Python code is not straightforward. In the context of Jinja templates, some existing attempts can be found:

- jinja_coverage is not maintained, though should give useful pointers
- django_coverage_plugin is another interesting take, though likely too specific to Django

Given the above, we will need to create our own coverage plugin suited to our needs. Initial research shows however that all the required information may not be easily accessed from the Jinja library. See:

- pallets/jinja#408
- pallets/jinja#674
- pallets/jinja#1130

Macros Testing

Important: This is not yet implemented.

Another aspect we can address with these tests is unit-testing of Jinja macros. This will ensure macros behaviour remains stable over time, and that their intent is clearly expressed in test cases.

To perform such unit-testing, one may approach it as follows:

```
from jinja2 import Environment

env = Environment(loader=FilesystemLoader('salt'))
macro_tpl = env.get_template('metalk8s/macro.sls')

# This is the exported `pkg_installed` macro
pkg_installed = macro_tpl.module.pkg_installed
```

Reference

Tests Configuration

The configuration of rendering tests can be found at `salt/tests/unit/formulas/config.yaml`, and is included below for reference:

```
---
# Default context options are listed here.
# Please be mindful of the number of cases generated, as these will apply to
# many formulas.
default_case:
  os: CentOS/7
  saltenv: metalk8s-2.8.0
```

(continues on next page)

(continued from previous page)

```
  minion_state: ready
  architecture: single-node
  volumes: none
  mode: minion

metalk8s:
  # Use the special `_skip` keyword to omit rendering of a directory or formula
  # _skip: true

  map.jinja:
    _cases:
      "CentOS 7":
        os: CentOS/7
      "RHEL 7":
        os: RedHat/7
      "RHEL 8":
        os: RedHat/8

  addons:
    dex:
      deployed:
        service-configuration.sls:
          _cases:
            "No service configuration (default)": {}
            "Existing service configuration (v1alpha2)":
              k8s_overrides:
                add:
                  - &dex_service_conf
                    kind: ConfigMap
                    apiVersion: v1
                    metadata:
                      name: metalk8s-dex-config
                      namespace: metalk8s-auth
                    data:
                      config.yaml: |-
                        apiVersion: addons.metalk8s.scality.com/v1alpha2
                        kind: DexConfig
                        spec: {}
            "Old service configuration (v1alpha1)":
              k8s_overrides:
                add:
                  - <<: *dex_service_conf
                    data:
                      config.yaml: |-
                        apiVersion: addons.metalk8s.scality.com/v1alpha1
                        kind: DexConfig
                        spec: {}
            "Unknown service configuration version":
              k8s_overrides:
                add:
                  - <<: *dex_service_conf
                    data:
```

(continues on next page)

(continued from previous page)

```
                    config.yaml: |-
                      apiVersion: addons.metalk8s.scality.com/v1
                      kind: DexConfig
                      spec: {}

  logging:
    loki:
      deployed:
        service-configuration.sls:
          _cases:
            "No existing configuration (default)": {}
            "Service configuration exists":
              k8s_overrides:
                add:
                  - kind: ConfigMap
                    apiVersion: v1
                    metadata:
                      name: metalk8s-loki-config
                      namespace: metalk8s-logging
                    data:
                      config.yaml: |-
                        apiVersion: addons.metalk8s.scality.com
                        kind: LokiConfig
                        spec: {}

  prometheus-operator:
    post-cleanup.sls:
      _cases:
        "No old rules (default)": {}
        "Old rules to remove":
          k8s_overrides:
            add:
              - kind: PrometheusRule
                apiVersion: monitoring.coreos.com/v1
                metadata:
                  name: example-old-rule
                  namespace: metalk8s-monitoring
                  labels:
                    app.kubernetes.io/part-of: metalk8s
                    # Assume our current version is higher than this
                    metalk8s.scality.com/version: "2.4.0"

    deployed:
      service-configuration.sls:
        _cases:
          "No existing configuration (default)": {}
          "Service configuration exists (only one of them)":
            k8s_overrides:
              add:
                - kind: ConfigMap
                  apiVersion: v1
                  metadata:
```

(continues on next page)

(continued from previous page)

```
                    name: metalk8s-prometheus-config
                    namespace: metalk8s-monitoring
                  data:
                    config.yaml: |-
                      apiVersion: addons.metalk8s.scality.com
                      kind: PrometheusConfig
                      spec: {}

  solutions:
    deployed:
      configmap.sls:
        _cases:
          "No solution available (default)": {}
          "Some solution available":
            pillar_overrides:
              metalk8s:
                solutions:
                  available:
                    example-solution:
                      - &example_solution
                        archive: >-
                          /srv/scality/releases/example-solution-1.0.0.iso
                        name: example-solution
                        version: "1.0.0"
                        id: example-solution-1.0.0
                        active: true
                        mountpoint: /srv/scality/example-solution-1.0.0

  nginx-ingress-control-plane:
    deployed:
      chart-deployment.sls:
        _cases:
          "Nominal (with soft antiAffinity in pillar)":
            pillar_overrides:
              networks:
                control_plane:
                  ingress:
                    controller:
                      replicas: 2
                      affinity:
                        podAntiAffinity:
                          soft:
                            - topologyKey: kubernetes.io/hostname

backup:
  deployed:
    secret-credentials.sls:
      _cases:
        "No existing Secret (default)": {}
        "Secret already exists":
          k8s_overrides:
            add:
```

(continues on next page)

(continued from previous page)

```
              - kind: Secret
                apiVersion: v1
                metadata:
                  name: backup-credentials
                  namespace: kube-system
                type: kubernetes.io/basic-auth
                data:
                  username: YmFja3Vw
                  password: UmpCVTdPRkM0NXpnYkk1Um9YUHR3eW5RcWxmZ3J2

container-engine:
  containerd:
    files:
      50-metalk8s.conf.j2:
        _cases:
          "From metalk8s.container-engine.containerd.installed":
            extra_context:
              containerd_args: [--log-level, info]
              environment:
                NO_PROXY: localhost,127.0.0.1,10.0.0.0/16
                HTTP_PROXY: http://my-proxy.local
                HTTPS_PROXY: https://my-proxy.local

kubernetes:
  apiserver-proxy:
    files:
      apiserver-proxy.yaml.j2:
        _cases:
          "From metalk8s.kubernetes.apiserver-proxy.installed":
            extra_context:
              image_name: >-
                metalk8s-registry-from-config.invalid/metalk8s-2.7.1/nginx:1.2.3
              config_digest: abcdefgh12345
              metalk8s_version: "2.7.1"

  coredns:
    deployed.sls:
      _cases:
        "Simple hostname soft anti-affinity (default)": {}
        "Multiple soft anti-affinity":
          pillar_overrides:
            kubernetes:
              coreDNS:
                affinity:
                  podAntiAffinity:
                    soft:
                      - topologyKey: kubernetes.io/hostname
                      - topologyKey: kubernetes.io/zone
                        weight: 10
        "Hard anti-affinity on hostname":
          pillar_overrides:
            kubernetes:
```

(continues on next page)

(continued from previous page)

```
              coreDNS:
                affinity:
                  podAntiAffinity:
                    hard:
                      - topologyKey: kubernetes.io/hostname
        "Multiple hard anti-affinity":
          pillar_overrides:
            kubernetes:
              coreDNS:
                affinity:
                  podAntiAffinity:
                    hard:
                      - topologyKey: kubernetes.io/hostname
                      - topologyKey: kubernetes.io/zone
        "Multiple hard and soft anti-affinity":
          pillar_overrides:
            kubernetes:
              coreDNS:
                affinity:
                  podAntiAffinity:
                    soft:
                      - topologyKey: kubernetes.io/hostname
                      - topologyKey: kubernetes.io/zone
                        weight: 10
                    hard:
                      - topologyKey: kubernetes.io/hostname
                      - topologyKey: kubernetes.io/zone

    files:
      coredns-deployment.yaml.j2:
        _cases:
          "From metalk8s.kubernetes.coredns.deployed":
            extra_context:
              replicas: 2
              label_selector:
                k8s-app: kube-dns
              affinity:
                podAntiAffinity:
                  preferredDuringSchedulingIgnoredDuringExecution:
                    - weight: 1
                      podAffinityTerm:
                        labelSelector:
                          matchLabels:
                            k8s-app: kube-dns
                        namespaces:
                          - kube-system
                        topologyKey: kubernetes.io/hostname

  etcd:
    files:
      manifest.yaml.j2:
        _cases:
```

(continues on next page)

(continued from previous page)

```
            "From metalk8s.kubernetes.etcd.installed":
              extra_context:
                name: etcd
                image_name: >-
                  metalk8s-registry-from-config.invalid/metalk8s-2.7.1/etcd:3.4.3
                command: [etcd, --some-arg, --some-more-args=toto]
                volumes:
                  - path: /var/lib/etcd
                    name: etcd-data
                  - path: /etc/kubernetes/pki/etcd
                    name: etcd-certs
                    readOnly: true
                etcd_healthcheck_cert: >-
                  /etc/kubernetes/pki/etcd/healthcheck-client.crt
                metalk8s_version: "2.7.1"
                config_digest: abcdefgh12345

    files:
      control-plane-manifest.yaml.j2:
        _cases:
          "From metalk8s.kubernetes.scheduler.installed":
            extra_context:
              name: kube-scheduler
              image_name: >-
                metalk8s-registry-from-config.invalid/metalk8s-2.7.1/kube-scheduler:1.18.
↪5
              host: "10.0.0.1"
              port: http-metrics
              scheme: HTTP
              command:
                - kube-scheduler
                - --address=10.0.0.1
                - --kubeconfig=/etc/kubernetes/scheduler.conf
                - --leader-elect=true
                - --v=0
              requested_cpu: 100m
              ports:
                - name: http-metrics
                  containerPort: 10251
              volumes:
                - path: /etc/kubernetes/scheduler.conf
                  name: kubeconfig
                  type: File
              metalk8s_version: "2.7.1"
              config_digest: abcdefgh12345

    kubelet:
      files:
        kubeadm.env.j2:
          _cases:
            "From metalk8s.kubernetes.kubelet.standalone":
              extra_context:
```

(continues on next page)

(continued from previous page)

```
              options:
                container-runtime: remote
                container-runtime-endpoint: >-
                  unix:///run/containerd/containerd.sock
                node-ip: "10.0.0.1"
                hostname-override: bootstrap
                v: 0

      service-systemd.conf.j2:
        _cases:
          "From metalk8s.kubernetes.kubelet.configured":
            extra_context:
              kubeconfig: /etc/kubernetes/kubelet.conf
              config_file: /var/lib/kubelet/config.yaml
              env_file: /var/lib/kubelet/kubeadm-flags.env

      service-standalone-systemd.conf.j2:
        _cases:
          "From metalk8s.kubernetes.kubelet.standalone":
            extra_context:
              env_file: /var/lib/kubelet/kubeadm-flags.env
              manifest_path: /etc/kubernetes/manifests

  mark-control-plane:
    files:
      bootstrap_node_update.yaml.j2.in:
        _cases:
          "From metalk8s.kubernetes.mark-control-plane.deployed":
            extra_context:
              node_name: bootstrap
              cri_socket: unix:///run/containerd/containerd.sock

    deployed.sls:
      _cases:
        "Default bootstrap target":
          pillar_overrides:
            bootstrap_id: bootstrap

node:
  grains.sls:
    _cases:
      "Grains are already set":
        minion_state: ready
      "Grains are not yet set":
        minion_state: new

orchestrate:
  apiserver.sls:
    _cases: &orch_base_cases
      "Single-node cluster":
        mode: master
        architecture: single-node
```

(continues on next page)

(continued from previous page)

```
      "Compact cluster": &orch_compact_arch
        mode: master
        architecture: compact

  backup:
    files:
      job.yaml.j2:
        _cases:
          "Create job manifest for a node":
            extra_context:
              node: master-1
              image: registry/some-image-name:tag

  register_etcd.sls:
    _cases:
      "Target a new master node": &orch_target_master_node
        <<: *orch_compact_arch
        pillar_overrides: &pillar_orch_target_master_node
          bootstrap_id: bootstrap
          orchestrate:
            node_name: master-1

  deploy_node.sls:
    _cases:
      "Target a new master node":
        <<: *orch_target_master_node
        _subcases:
          "Minion already exists (default)": {}
          "Minion is new":
            minion_state: new
          "Skip drain":
            pillar_overrides:
              <<: *pillar_orch_target_master_node
              orchestrate:
                node_name: master-1
                skip_draining: true
          "Change drain timeout":
            pillar_overrides:
              <<: *pillar_orch_target_master_node
              orchestrate:
                node_name: master-1
                drain_timeout: 60
          "Skip etcd role":
            pillar_overrides:
              <<: *pillar_orch_target_master_node
              metalk8s:
                nodes:
                  master-1:
                    skip_roles: [etcd]

  etcd.sls:
    _cases: *orch_base_cases
```

(continues on next page)

(continued from previous page)

```
bootstrap:
  accept-minion.sls:
    _cases:
      "From master":
        mode: master
        _subcases:
          "Bootstrap minion is available":
            pillar_overrides:
              bootstrap_id: bootstrap
          "Bootstrap minion is unavailable":
            # The mocks will only return known minions for `manage.up`
            pillar_overrides:
              bootstrap_id: unavailable-bootstrap

  init.sls:
    _cases:
      "From master":
        mode: master
        _subcases:
          # FIXME: metalk8s.orchestrate.bootstrap.init does not handle an
          # unavailable minion yet
          "Bootstrap node exists in K8s":
            pillar_overrides:
              bootstrap_id: bootstrap
          "Bootstrap node does not exist yet":
            pillar_overrides:
              bootstrap_id: bootstrap
              metalk8s:
                nodes: {}

certs:
  renew.sls:
    _cases:
      "From master":
        mode: master
        _subcases:
          "No certificate to process":
            pillar_overrides:
              orchestrate:
                certificates: []
                target: bootstrap
          "Some certificates to process":
            pillar_overrides:
              orchestrate:
                certificates:
                  # Client
                  - /etc/kubernetes/pki/etcd/salt-master-etcd-client.crt
                  # Kubeconfig
                  - /etc/kubernetes/admin.conf
                  # Server
                  - /etc/kubernetes/pki/apiserver.crt
```

(continues on next page)

(continued from previous page)

```
                target: bootstrap

  downgrade:
    init.sls:
      _cases:
        "Single node cluster":
          architecture: single-node
          _subcases:
            "Destination matches (default)": {}
            "Destination is lower than node version":
              pillar_overrides:
                metalk8s:
                  cluster_version: 2.7.3

        "Compact cluster":
          architecture: compact
          _subcases: &downgrade_subcases
            "Destination matches (default)": {}
            "Destination is lower than all nodes":
              pillar_overrides:
                metalk8s:
                  cluster_version: 2.7.3
            "Destination is lower than some nodes":
              pillar_overrides:
                metalk8s:
                  cluster_version: 2.7.3
                  nodes:
                    master-1:
                      version: 2.7.3

        "Standard cluster":
          architecture: standard
          _subcases: *downgrade_subcases

        "Extended cluster":
          architecture: extended
          _subcases: *downgrade_subcases

    precheck.sls:
      _cases:
        "Saltenv matches highest node version (default)":
          saltenv: metalk8s-2.8.0
          # Use multi-node to verify handling of heterogeneous versions
          architecture: compact
          _subcases:
            "All nodes already in desired version (default)": {}
            "Desired version is too old":
              pillar_overrides:
                metalk8s:
                  cluster_version: 2.6.1
            "Some nodes in desired version":
              pillar_overrides:
```

(continues on next page)

(continued from previous page)

```
                metalk8s:
                  cluster_version: 2.7.3
                  nodes:
                    master-1:
                      version: 2.7.3
            "Some nodes in older version than desired":
              pillar_overrides:
                metalk8s:
                  cluster_version: 2.7.3
                  nodes:
                    master-1:
                      version: 2.7.1
            "Some node is not ready":
              k8s_overrides: &k8s_patch_node_not_ready
                edit:
                  - apiVersion: v1
                    kind: Node
                    metadata:
                      name: master-1
                    status:
                      conditions:
                        - type: Ready
                          status: false
                          reason: NodeHasDiskPressure

        "Saltenv is higher than highest node version":
          saltenv: metalk8s-2.9.0
          architecture: single-node
          pillar_overrides:
            metalk8s:
              cluster_version: 2.7.3
              nodes:
                bootstrap:
                  version: 2.8.0

        "Saltenv is lower than highest node version":
          saltenv: metalk8s-2.7.3
          architecture: single-node
          pillar_overrides:
            metalk8s:
              cluster_version: 2.7.3
              nodes:
                bootstrap:
                  version: 2.8.0

    solutions:
      deploy-components.sls:
        _cases:
          "Empty SolutionsConfig (default)": {}
          "Errors in solutions pillar":
            pillar_overrides:
              metalk8s:
```

(continues on next page)

(continued from previous page)

```
            solutions:
              _errors: ["Some error"]

        "Specific Solution version to deploy":
          pillar_overrides:
            bootstrap_id: bootstrap
            metalk8s:
              solutions:
                available: &base_example_available_solution
                  example-solution:
                    - id: example-solution-1.2.3
                      version: "1.2.3"
                      name: example-solution
                      display_name: Example Solution
                      mountpoint: /srv/scality/example-solution-1.2.3
                      manifest:
                        spec:
                          operator:
                            image:
                              name: example-operator
                              tag: "1.2.3"
                          images:
                            example-operator: "1.2.3"
                config:
                  active:
                    example-solution: "1.2.3"

        "Latest Solution version to deploy":
          pillar_overrides:
            bootstrap_id: bootstrap
            metalk8s:
              solutions:
                available: *base_example_available_solution
                config:
                  active:
                    example-solution: "latest"

        "Some available Solution to remove":
          pillar_overrides:
            bootstrap_id: bootstrap
            metalk8s:
              solutions:
                available: *base_example_available_solution
                config:
                  active: {}

        "Some desired Solution name is not available":
          pillar_overrides:
            metalk8s:
              solutions:
                available: *base_example_available_solution
                config:
```

(continues on next page)

(continued from previous page)

```
                active:
                  unknown-solution: "1.2.3"

      "Some desired Solution version is not available":
        pillar_overrides:
          metalk8s:
            solutions:
              available: *base_example_available_solution
              config:
                active:
                  example-solution: "4.5.6"

  import-components.sls:
    _cases:
      "Target an existing Bootstrap node":
        pillar_overrides:
          bootstrap_id: bootstrap

  prepare-environment.sls:
    _cases:
      "Environment does not exist (default)":
        pillar_overrides: &base_pillar_prepare_environment
          orchestrate:
            env_name: example-env

      "Errors in pillar":
        pillar_overrides:
          <<: *base_pillar_prepare_environment
          metalk8s:
            solutions:
              # FIXME: likely this formula should only look at
              # pillar.metalk8s.solutions.environments._errors
              _errors: ["Some error", "Some other error"]
              environments:
                _errors: ["Some error"]

  files:
    operator:
      service_account.yaml.j2:
        _cases:
          "Example Solution v1.2.3 (see ../../prepare-environment.sls)":
            extra_context: &base_context_solution_operator_files
              solution: example-solution
              version: "1.2.3"
              namespace: example-env

      configmap.yaml.j2:
        _cases:
          "Example Solution v1.2.3 (see ../../prepare-environment.sls)":
            extra_context:
              <<: *base_context_solution_operator_files
              registry: metalk8s-registry-from-config.invalid
```

(continues on next page)

(continued from previous page)

```
      deployment.yaml.j2:
        _cases:
          "Example Solution v1.2.3 (see ../../prepare-environment.sls)":
            extra_context:
              <<: *base_context_solution_operator_files
              repository: >-
                metalk8s-registry-from-config.invalid/example-solution-1.2.3
              image_name: example-operator
              image_tag: "1.2.3"

      role_binding.yaml.j2:
        _cases:
          "Example Solution v1.2.3 (see ../../prepare-environment.sls)":
            extra_context:
              <<: *base_context_solution_operator_files
              role_kind: ClusterRole
              role_name: example-role

  upgrade:
    init.sls:
      _cases:
        "Single node cluster":
          architecture: single-node
          _subcases:
            "Destination matches (default)": {}
            "Destination is higher than node version":
              pillar_overrides:
                metalk8s:
                  cluster_version: 2.9.0

        "Compact cluster":
          architecture: compact
          _subcases: &upgrade_subcases
            "Destination matches (default)": {}
            "Destination is higher than all nodes":
              pillar_overrides:
                metalk8s:
                  cluster_version: 2.9.0
            "Destination is higher than some nodes":
              pillar_overrides:
                metalk8s:
                  cluster_version: 2.9.0
                  nodes:
                    master-1:
                      version: 2.9.0

        "Standard cluster":
          architecture: standard
          _subcases: *upgrade_subcases

        "Extended cluster":
```

(continues on next page)

(continued from previous page)

```
            architecture: extended
            _subcases: *upgrade_subcases

      precheck.sls:
        _cases:
          "Saltenv matches destination version (default)":
            saltenv: metalk8s-2.8.0
            # Use multi-node to verify handling of heterogeneous versions
            architecture: compact
            _subcases:
              "All nodes already in desired version (default)": {}
              "All nodes in older compatible version":
                pillar_overrides:
                  metalk8s:
                    nodes: &_upgrade_compatible_nodes
                      bootstrap: &_upgrade_compatible_node_version
                        version: 2.7.3
                      master-1: *_upgrade_compatible_node_version
                      master-2: *_upgrade_compatible_node_version
              "Current version of some node is too old":
                pillar_overrides:
                  metalk8s:
                    nodes:
                      <<: *_upgrade_compatible_nodes
                      master-1:
                        version: 2.6.1
              "Current version of some node is newer than destination":
                pillar_overrides:
                  metalk8s:
                    nodes:
                      <<: *_upgrade_compatible_nodes
                      master-1:
                        version: 2.9.0
              "Some nodes in older version than desired":
                pillar_overrides:
                  metalk8s:
                    cluster_version: 2.7.3
                    nodes:
                      master-1:
                        version: 2.7.1
              "Some node is not ready":
                k8s_overrides: *k8s_patch_node_not_ready

          "Saltenv does not match destination version":
            saltenv: metalk8s-2.7.3
            architecture: single-node
            pillar_overrides:
              metalk8s:
                cluster_version: 2.8.0

  reactor:
    certs:
```

(continues on next page)

(continued from previous page)

```
    renew.sls.in:
      _cases:
        "Sample beacon event":
          extra_context:
            data:
              id: bootstrap
              certificates:
                - cert_path: /path/to/cert.pem
                - cert_path: /path/to/other.pem

repo:
  files:
    metalk8s-registry-config.inc.j2:
      _cases:
        "From metalk8s.repo.configured":
          extra_context:
            archives: &example_archives
              metalk8s-2.7.1:
                iso: /archives/metalk8s.iso
                path: /srv/scality/metalk8s-2.7.1
                version: "2.7.1"

    nginx.conf.j2:
      _cases:
        "From metalk8s.repo.configured":
          extra_context:
            listening_address: "10.0.0.1"
            listening_port: 8080

    repositories-manifest.yaml.j2:
      _cases:
        "From metalk8s.repo.installed":
          extra_context:
            container_port: 8080
            image: >-
              metalk8s-registry-from-config.invalid/metalk8s-2.7.1/nginx:1.2.3
            name: repositories
            version: "1.0.0"
            archives: *example_archives
            solutions: {}
            package_path: /packages
            image_path: /images/
            nginx_confd_path: /var/lib/metalk8s/repositories/conf.d
            probe_host: "10.0.0.1"
            metalk8s_version: "2.7.1"
            config_digest: abcdefgh12345

  redhat.sls:
    _cases:
      "CentOS 7":
        os: CentOS/7
      "RHEL 7":
```

(continues on next page)

(continued from previous page)

```
        os: RedHat/7
      "RHEL 8":
        os: RedHat/8

salt:
  master:
    certs:
      salt-api.sls:
        _cases:
          "Minion is standalone (bootstrap)":
            minion_state: standalone
          "Minion is connected to master and CA (default)":
            minion_state: ready

    files:
      master-99-metalk8s.conf.j2:
        _cases:
          "From metalk8s.salt.master.configured":
            extra_context:
              debug: true
              salt_ip: "10.0.0.1"
              kubeconfig: /etc/salt/master-kubeconfig.conf
              salt_api_ssl_crt: /etc/salt/pki/api/salt-api.crt
              saltenv: metalk8s-2.7.1

      salt-master-manifest.yaml.j2:
        _cases:
          "From metalk8s.salt.master.installed":
            extra_context:
              debug: true
              image: salt-master
              version: "3002.2"
              archives:
                metalk8s-2.7.1:
                  path: /srv/scality/metalk8s-2.7.1
                  iso: /archives/metalk8s-2.7.1
                  version: "2.7.1"
              solution_archives:
                example-solution-1-2-3: /srv/scality/example-solution-1.2.3
              salt_ip: "10.0.0.1"
              config_digest: abcdefgh12345
              metalk8s_version: "2.7.1"

  minion:
    files:
      minion-99-metalk8s.conf.j2:
        _cases:
          "From metalk8s.salt.minion.configured":
            extra_context:
              debug: true
              master_hostname: "10.233.0.123"
              minion_id: bootstrap
```

(continues on next page)

(continued from previous page)

```
              saltenv: metalk8s-2.7.1

solutions:
  available.sls:
    _cases:
      # See ./data/base_pillar.yaml
      "Empty config (default)": {}
      "Initial state (errors)":
        pillar_overrides:
          metalk8s:
            solutions:
              _errors: [Cannot read config file]
              available: {}
              config:
                _errors: [Cannot read config file]

      "New archive in config":
        pillar_overrides:
          metalk8s:
            solutions:
              available:
                example-solution:
                  - *example_solution
              config:
                archives:
                  - /srv/scality/releases/example-solution-1.0.0.iso
                  - /srv/scality/releases/example-solution-1.2.0.iso

      "Active archive removed from config":
        pillar_overrides:
          metalk8s:
            solutions:
              available:
                example-solution:
                  - *example_solution
              config:
                archives: []

      "Inactive archive removed from config":
        pillar_overrides:
          metalk8s:
            solutions:
              available:
                example-solution:
                  - *example_solution
                  - archive: >-
                      /srv/scality/releases/example-solution-1.2.0.iso
                    name: example-solution
                    version: "1.2.0"
                    id: example-solution-1.2.0
                    active: false
                    mountpoint: /srv/scality/example-solution-1.2.0
```

(continues on next page)

(continued from previous page)

```
              config:
                archives:
                  - /srv/scality/releases/example-solution-1.0.0.iso

utils:
  httpd-tools:
    installed.sls:
      _cases:
        "RedHat family":
          os: CentOS/7

volumes:
  unprepared.sls:
    _cases: &volumes_cases
      "No volume (default)":
        volumes: none
        _subcases: &volumes_subcases
          "No target volume (default)": {}
          "Target a single volume":
            pillar_overrides:
              volume: bootstrap-prometheus
      "Errors in volumes pillar":
        volumes: errors
        _subcases: *volumes_subcases
      "Only sparse loop volumes":
        volumes: sparse
        _subcases: *volumes_subcases
      "Only raw block volumes":
        volumes: block
        _subcases: *volumes_subcases
      "Mix of sparse and block volumes":
        volumes: mix
        _subcases: *volumes_subcases

  prepared.sls:
    _cases:
      <<: *volumes_cases
      "Volumes pillar is set to None (bootstrap)":
        volumes: bootstrap
```

3.5.2 Development Best Practices

Commit Best Practices

Pre-commit hooks

Some pre-commit hooks are defined to do some linting checks and also to format all Python code automatically.

Those checks are also run in the CI pre-merge test suite to enforce code linting.

To enable pre-commit hook to run automatically when committing, install it as follows:

```
pip install pre-commit
pre-commit install
```

You can skip this pre-commit hook on a specific commit `git commit --no-verify`.

To run pre-commit manually, use tox:

```
tox -e pre-commit
```

It is also possible to run only a specific hook (e.g. for pylint `tox -e pre-commit pylint`).

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.

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
```

(continues on next page)

(continued from previous page)

```
  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.

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.

Python best practices

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

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
        # [...]
```

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

3.6 Integrating with MetalK8s

3.6.1 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.

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

For more details, see the following documentation pages:

- *Solution archive guidelines*

- *Solution Operator guidelines*
- (TODO) Solution UI guidelines

Once a Solution is imported in MetalK8s, a user can deploy one or more versions of the Solution Operator, using either the MetalK8s Solution CLI (`./solutions.sh`) or the MetalK8s UI Environment page, into separate `Environments` (namespaces). Using the Operator-defined `CustomResource(s)`, the user can then effectively deploy the application packaged in the Solution.

How is a *Solution* declared in MetalK8s?

MetalK8s 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 use a `SolutionsConfiguration` object, stored in `/etc/metalk8s/solutions.yaml`, to declare which Solutions are available to the cluster, from the bootstrap node.

Here is how it looks like:

```
apiVersion: metalk8s.scality.com/v1alpha1
kind: SolutionsConfiguration
archives:
  - /solutions/storage_1.0.0.iso
  - /solutions/storage_latest.iso
  - /other_solutions/computing.iso
active:
  storage: 1.0.0
```

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

Note: Removing an archive path from the `Solutions` list effectively removes its related resources (CRDs, images) from a MetalK8s cluster.

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 Operator

MAY:

- 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)

SHOULD:

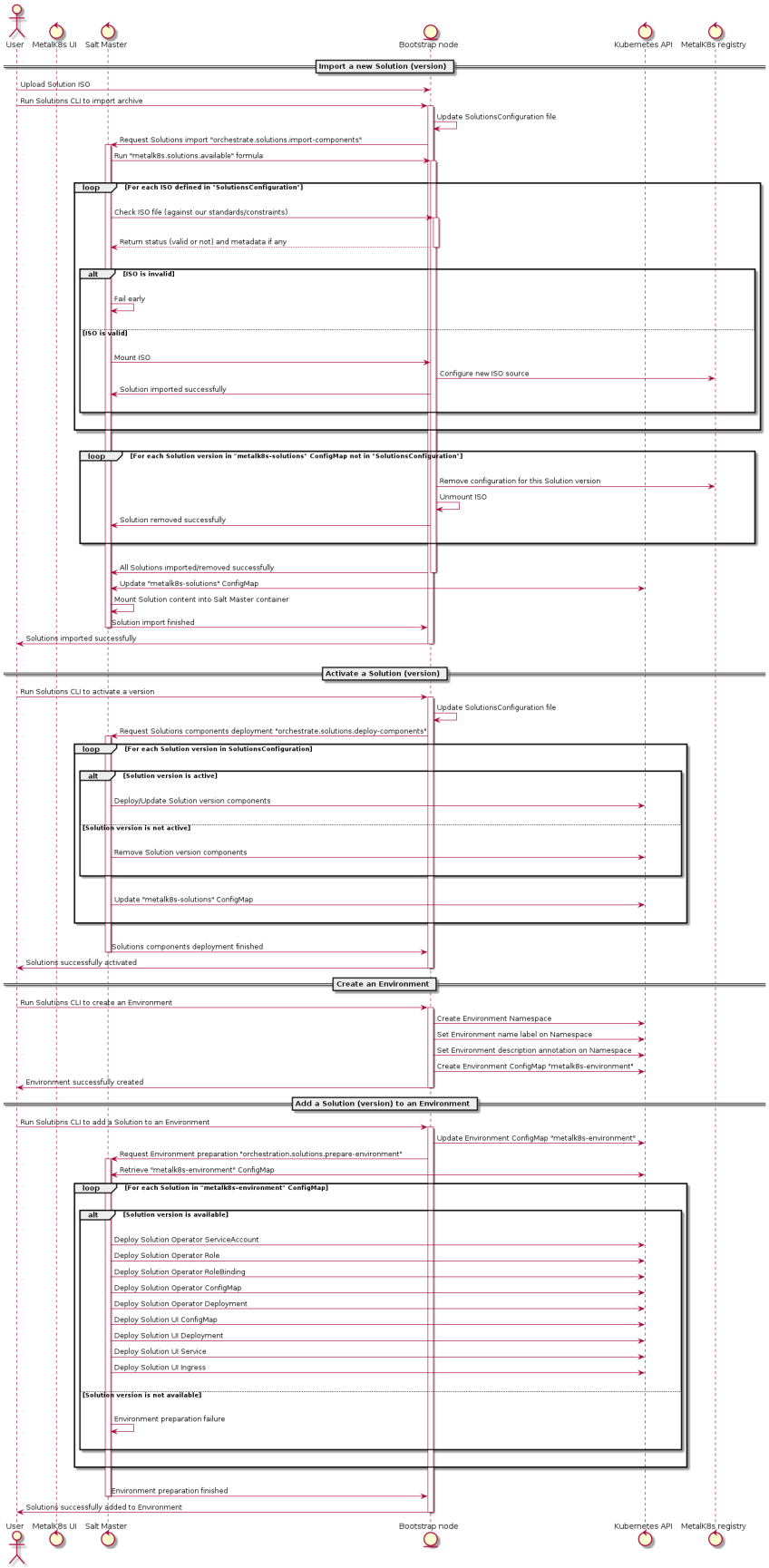
- Use MetalK8s monitoring services (Prometheus and Grafana)

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*.

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.


User
MetalK8s UI
Salt Master
Bootstrap node
Kubernetes API
MetalK8s registry
Import a new Solution (version)
Upload Solution ISO
Run Solutions CLI to import archive
Update SolutionsConfiguration file
Request Solutions import "orchestrate.solutions.import-components"
Run "metalk8s.solutions.available" formula
loop
[For each ISO defined in "SolutionsConfiguration"]
Check ISO file (against our standards/constraints)
Return status (valid or not) and metadata if any
alt
[ISO is invalid]
Fail early
[ISO is valid]
Mount ISO
Configure new ISO source
Solution imported successfully
loop
[For each Solution version in "metalk8s-solutions" ConfigMap not in "SolutionsConfiguration"]
Remove configuration for this Solution version
Unmount ISO
Solution removed successfully
All Solutions imported/removed successfully
Update "metalk8s-solutions" ConfigMap
Mount Solution content into Salt Master container
Solution import finished
Solutions imported successfully
Activate a Solution (version)
Run Solutions CLI to activate a version
Update SolutionsConfiguration file
Request Solutions components deployment "orchestrate.solutions.deploy-components"
loop
[For each Solution version in SolutionsConfiguration]
alt
[Solution version is active]
Deploy/Update Solution version components
[Solution version is not active]
Remove Solution version components
Update "metalk8s-solutions" ConfigMap
Solutions components deployment finished
Solutions successfully activated
Create an Environment
Run Solutions CLI to create an Environment
Create Environment Namespace
Set Environment name label on Namespace
Set Environment description annotation on Namespace
Create Environment ConfigMap "metalk8s-environment"
Environment successfully created
Add a Solution (version) to an Environment
Run Solutions CLI to add a Solution to an Environment
Update Environment ConfigMap "metalk8s-environment"
Request Environment preparation "orchestration.solutions.prepare-environment"
Retrieve "metalk8s-environment" ConfigMap
loop
[For each Solution in "metalk8s-environment" ConfigMap]
alt
[Solution version is available]
Deploy Solution Operator ServiceAccount
Deploy Solution Operator Role
Deploy Solution Operator RoleBinding
Deploy Solution Operator ConfigMap
Deploy Solution Operator Deployment
Deploy Solution UI ConfigMap
Deploy Solution UI Deployment
Deploy Solution UI Service
Deploy Solution UI Ingress
[Solution version is not available]
Environment preparation failure
Environment preparation finished
Solutions successfully added to Environment

3.6.2 Solution archive guidelines

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

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}`.

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/
```

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
├── manifest.yaml
├── operator
│   └── deploy
│       ├── crds
│       │   └── some_crd_name.yaml
│       └── role.yaml
└── registry-config.inc
```

Product information

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

It must respect the following format (currently `solutions.metalk8s.scality.com/v1alpha1`, as specified by the `apiVersion` value):

```
apiVersion: solutions.metalk8s.scality.com/v1alpha1
kind: Solution
metadata:
  annotations:
    solutions.metalk8s.scality.com/display-name: Solution Name
  labels: {}
  name: solution-name
spec:
  images:
    - some-extra-image:2.0.0
    - solution-name-operator:1.0.0
    - solution-name-ui:1.0.0
  operator:
    image:
      name: solution-name-operator
      tag: 1.0.0
  version: 1.0.0
```

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

```
solutions.metalk8s.scality.com/build-timestamp: \
  $(date -u +%Y-%m-%dT%H:%M:%SZ)
solutions.metalk8s.scality.com/git-revision: \
  $(git describe --always --long --tags --dirty)
```

A simple script to generate this manifest can be found in MetalK8s repository *examples/metalk8s-solution-example/manifest.py*, use it as follows:

```
./manifest.py --name "example-solution" \
    --annotation "solutions.metalk8s.scality.com/build-timestamp" \
    "$(date -u +%Y-%m-%dT%H:%M:%SZ)" \
    --annotation "solutions.metalk8s.scality.com/build-host" "$(hostname)" \
    --annotation "solutions.metalk8s.scality.com/development-release" "1" \
    --annotation "solutions.metalk8s.scality.com/display-name" "Example Solution" \
    --annotation "solutions.metalk8s.scality.com/git-revision" \
    "$(git describe --always --long --tags --dirty)" \
    --extra-image "base-server" "0.1.0-dev" \
    --operator-image "example-solution-operator" "0.1.0-dev" \
    --ui-image "example-solution-ui" "0.1.0-dev" \
    --version "0.1.0-dev"
```

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, images can be saved 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
$ # 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
```

The `images` directory should now resemble this:

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

Once all the images are stored this way, de-duplication of layers can be done with hardlinks, using the tool hardlink:

```
$ hardlink -c images
```

A detailed procedure for generating the expected layout is available at NicolasT/static-container-registry. The script provided there, or the one vendored in this repository (located at `buildchain/static-container-registry`) can be used 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 `<metadata:name>-<spec:version>` from *Product information*, and will be mounted at `/srv/scality/<metadata:name>-<spec:version>`.

Warning: Operators should not rely on this naming pattern for finding the images for their resources. Instead, the full repository endpoints will be exposed to the Operator container through a configuration file passed to the operator binary. 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*.

Operator

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

Web UI

3.6.3 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.

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.yaml
    └── role.yaml
```

Most of these files are generated when using the Operator SDK.

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.

Configuration

Solution Operator must implement a `--config` option which will be used by MetalK8s to provide various useful information needed by the Operator, such as the endpoints for the container images. The given configuration looks like:

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

In example, for an online installation without MetalK8s providing the repository, this configuration could be:

```
apiVersion: solutions.metalk8s.scality.com/v1alpha1
kind: OperatorConfig
repositories:
  1.0.0:
    - endpoint: registry.scality.com/zenko
      images:
        - cloudserver:1.0.0
        - zenko-quorum:1.0.0
    - endpoint: quay.io/coreos
      images:
        - prometheus-operator:v0.34.0
```

This configuration allows the Operator to retrieve dynamically where the container images are stored for each version of a given Solution.

Roles

Solution must ship a `role.yaml` file located in `/operator/deploy` directory. This file is a manifest which declares all necessary `Role` and `ClusterRole` objects needed by the Operator. MetalK8s will take care of deploying these objects, create a `ServiceAccount` named `<solution_name>-operator` and all needed `RoleBinding` to bind these roles to this account.

> **Warning:** Only `Role` and `ClusterRole` kinds are allowed in this file, the deployment of the Solution fails if any other resource is found.

3.6.4 Deploying And Experimenting

Given the solution ISO is correctly generated, a script utiliy has been added to manage Solutions. This script is located at the root of Metalk8s archive:

```
/srv/scality/metalk8s-2.11.7/solutions.sh
```

Import a Solution

Importing a Solution will mount its ISO and expose its container images.

To import a Solution into MetalK8s cluster, use the `import` subcommand:

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

The `--archive` option can be provided multiple times to import several Solutions ISOs at the same time:

```
./solutions.sh import --archive </path/to/solution1.iso> \
  --archive </path/to/solution2.iso>
```

Unimport a Solution

To unimport a Solution from MetalK8s cluster, use the `unimport` subcommand:

> **Warning:** Images of a Solution will no longer be available after an archive removal

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

Activate a Solution

Activating a Solution version will deploy its CRDs.

To activate a Solution in MetalK8s cluster, use the `activate` subcommand:

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

Deactivate a Solution

To deactivate a Solution from Metalk8s cluster, use the `deactivate` subcommand:

```
./solutions.sh deactivate --name <solution-name>
```

Create an Environment

To create a Solution Environment, use the `create-env` subcommand:

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

By default, it will create a Namespace named after the `<environment-name>`, but it can be changed, using the `--namespace` option:

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

It's also possible to use the previous command to create multiple Namespaces (one at a time) in this Environment, allowing Solutions to run in different Namespaces.

Delete an Environment

To delete an Environment, use the `delete-env` subcommand:

> **Warning:** This will destroy everything in the said Environment, with no way back

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

In case of multiple Namespaces inside an Environment, it's also possible to only delete a single Namespace, using:

```
./solutions.sh delete-env --name <environment-name> \
  --namespace <namespace-name>
```

Add a Solution in an Environment

Adding a Solution will deploy its Operator in the Environment.

To add a Solution in an Environment, use the `add-solution` subcommand:

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

In case of non-default Namespace (not corresponding to `<environment-name>`) or multiple Namespaces in an Environment, Namespace in which the Solution will be added must be precised, using the `--namespace` option:

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

Delete a Solution from an Environment

To delete a Solution from an Environment, use the `delete-solution` subcommand:

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

Upgrade/Downgrade a Solution

Before starting, the destination version must have been imported.

Patch the Environment ConfigMap, with the destination version:

```
kubectl patch cm metalk8s-environment --namespace <namespace-name> \
  --patch '{"data": {"<solution-name>": "<solution-version-dest>"}}'
```

Apply the change with Salt:

```
salt_container=$(
  crictl ps -q \
  --label io.kubernetes.pod.namespace=kube-system \
  --label io.kubernetes.container.name=salt-master \
  --state Running
)
crictl exec -i "$salt_container" salt-run state.orchestrate \
  metalk8s.orchestrate.solutions.prepare-environment \
  pillar="{'orchestrate': {'env_name': '<environment-name>'}}"
```

3.7 Shared Tooling

As part of the MetalK8s project, some re-usable tooling was defined, and is documented here for interested readers.

3.7.1 `lib_alert_tree` Python library

This Python library serves as a common ground when composing Prometheus alerts into hierarchical trees in a form that can be best consumed by user interfaces and monitoring tools.

It features an optional helper to generate a command-line tool which helps with discovery of an existing tree, and provides an easy command to render a corresponding *PrometheusRule* manifest.

Overview

The library only has a handful of modules:

lib_alert_tree.models

The main module you will use to build the desired hierarchy of alerts, through the use of `ExistingAlert` and `DerivedAlert`.

See the reference.

lib_alert_tree.prometheus

Holds the data containers used to render accurate Prometheus configuration.

See the reference.

lib_alert_tree.kubernetes

Provides some helpers for listing common existing alerts for some usual K8s objects (e.g. `deployment_alerts` will list all alerts for a *Deployment*, given its name and namespace).

See the reference.

lib_alert_tree.cli

Exposes the `generate_cli` method, which can be used to define a Click entrypoint, with helpful subcommands for interacting with one or more alert trees.

See the reference.

Topics

Installation

Installing as a `git` dependency

> **Warning:** While this approach is likely the simplest, it requires a full clone of the `scality/metalk8s` repository, which is far from optimal.

Important: This requires `pip` version 19 or higher to handle PEP 517 packages as generated by Poetry.

```
$ python3 -m pip install --use-pep517 \
    "git+https://github.com/scality/metalk8s.git#egg=lib_alert_tree&subdirectory=tools/
↪lib-alert-tree"
```

Installing for development with Poetry

Important: The command below needs to be executed from `tools/lib-alert-tree/`

```
$ poetry install
```

Installing from a wheel or a source distribution

Important: The commands below need to be executed from `tools/lib-alert-tree/`

Packages are not currently published to any remote repository.

Use `poetry build` to generate the following artifacts:

```
./dist/
├── lib_alert_tree-0.1.0-py3-none-any.whl
└── lib-alert-tree-0.1.0.tar.gz
```

The wheel can be installed with:

```
$ pip install ./dist/lib_alert_tree-0.1.0-py3-none-any.whl
Processing ./dist/lib_alert_tree-0.1.0-py3-none-any.whl
Installing collected packages: lib-alert-tree
Successfully installed lib-alert-tree-0.1.0
```

Similarly, the source can be installed with:

```
$ pip install ./dist/lib-alert-tree-0.1.0.tar.gz
Processing ./dist/lib-alert-tree-0.1.0.tar.gz
  Installing build dependencies ... done
Building wheels for collected packages: lib-alert-tree
  Running setup.py bdist_wheel for lib-alert-tree ... done
  Stored in directory: /home/user/.cache/pip/wheels/a8/34/5d/
↪893ab01bce6a9f8fdbbbc5d4615dfd34e928638f22e7131759
Successfully built lib-alert-tree
Installing collected packages: lib-alert-tree
Successfully installed lib-alert-tree-0.1.0
```

Usage

Make sure you have *installed the library* and its dependencies already (enable the `cli` extra).

Create a simple Python module called `example.py`, and paste the following contents (or download it `from here`):

```
from lib_alert_tree.models import DerivedAlert as Derived
from lib_alert_tree.models import ExistingAlert as Existing
from lib_alert_tree.models import Relationship
from lib_alert_tree.models import severity_pair
```

(continues on next page)

(continued from previous page)

```
from lib_alert_tree.cli import generate_cli

# Existing alerts are defined with their name and labels only.
FOO_WARNING = Existing("FooServiceTooManyErrors", severity="warning")
FOO_CRITICAL = Existing("FooServiceTooManyErrors", severity="critical")

# There are shortcuts for defining the severity.
BAR_BAZ_WARNING = Existing.warning("BarAlmostOutOfSpace", bar="baz")
BAR_BAZ_CRITICAL = Existing.critical("BarAlmostOutOfSpace", bar="baz")

# Derived alerts are built from a list of children and a relationship type.
# To be serialized into valid Prometheus configuration, some other attributes are
# required.
FOOBAR_WARNING = Derived.warning(
    "FooBarDegraded",
    relationship=Relationship.ANY,
    children=[FOO_WARNING, BAR_BAZ_WARNING],
    duration="1m",
    summary="The FooBar service is degraded.",
)
FOOBAR_CRITICAL = Derived.critical(
    "FooBarAtRisk",
    relationship=Relationship.ALL,
    children=[FOO_CRITICAL, BAR_BAZ_CRITICAL],
    duration="1m",
    summary="The FooBar service is at risk.",
)

# The above "pair" is very common, so we built a shortcut for it.
ROOT_WARNING, ROOT_CRITICAL = severity_pair(
    "Example",
    summary_name="The example app",
    relationship=Relationship.ANY,
    warning_children=[FOOBAR_WARNING, Existing.warning("QuxNotProgressing")],
    critical_children=[FOOBAR_CRITICAL, Existing.critical("QuxNotProgressing")],
    duration="5m",
)

main = generate_cli(
    roots={"warning": ROOT_WARNING, "critical": ROOT_CRITICAL},
    prometheus_rule_labels={"metalk8s.scality.com/monitor": ""},
)

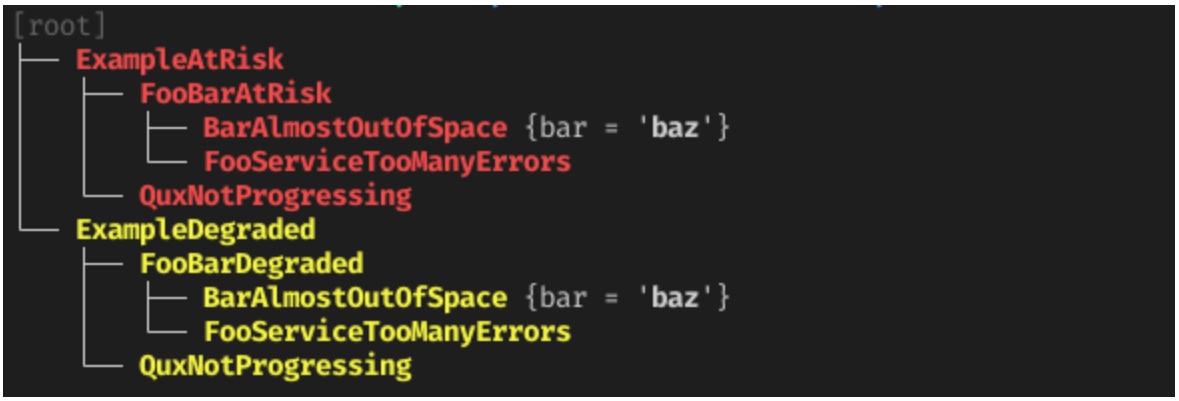
if __name__ == "__main__":
    main()
```

Read through the comments for an explanation of the basic features of this library.

Now, simply run the following to display the tree of alerts you just defined:

```
$ python -m example show
```

This will print out to the console:



You can also generate a *PrometheusRule* manifest using:

```
$ python -m example gen-rule \
    --name example.rules --namespace example \
    --out example_rule.yaml
```

The generated manifest should look like `this one`. Note that only `DerivedAlert` instances are part of this manifest (exiting alerts should not need any additional rules, and are only referred to in the derived `expr`).

Use the `--help` option to print a help message and discover more functionalities provided by these commands.

Going Further

To see a more advanced example, have a look at the MetalK8s alert tree defined under `tools/lib-alert-tree/metalk8s`. An easy access is provided through a `tox` environment:

```
$ tox -e alert-tree -- show
```

Reference

lib_alert_tree

Library for creating and manipulating hierarchies of Prometheus alerts.

lib_alert_tree.__version__ = '0.1.0'
The version of this project.

lib_alert_tree.models

Classes for manipulating tree-like hierarchies of alerts.

class lib_alert_tree.models.**BaseAlert**
Base-class for alert models.

abstract property alert_rule
Generate a 'prometheus.AlertRule' corresponding to this model.

create_node(*tree*, *parent=None*)
Add this alert to a tree.

classmethod critical(*name*, ***kwargs*)
Helper to generate an alert with "critical" severity.

property json_path
A JSON Path filter to retrieve the alert represented by this model.

property pretty_str
A colored string (based on severity), if Click is available.

property query
A PromQL query to retrieve the alert represented by this model.

classmethod warning(*name*, ***kwargs*)
Helper to generate an alert with "warning" severity.

class lib_alert_tree.models.**DerivedAlert**(*name*, *children*, *relationship*, *group_by=None*, ***params*)
A non-leaf node (logical) in a tree of alerts.

This is a computed alert rule, which maps a list of weighted edges towards other nodes (leaves or not) with a single alert.

In practice, one can fully build up a tree using a *DerivedAlert* instance as the root, using the *build_tree* method.

property alert_rule
Generate a 'prometheus.AlertRule' corresponding to this model.

build_rules_group(*name*)
Build a 'prometheus.RulesGroup' from all linked DerivedAlerts.

build_tree(*parent=None*, *tree=None*)
Build a 'treelib.Tree' object by recursing through children.

class lib_alert_tree.models.**ExistingAlert**(*name*, *severity=None*, ***labels*)
A leaf in a tree of logical alerts.

In essence, this represents an already existing alert rule in some Prometheus configuration, from which we will build composition rules. It only stores a name and a sufficient set of labels for querying.

property alert_rule
Generate a 'prometheus.AlertRule' corresponding to this model.

class lib_alert_tree.models.**Relationship**(*value*)
The type of relationship a parent has towards its children.

static build_json_path(*children*, *group_by=None*)
Build a JSON Path to retrieve alert children.

build_query(*children*, *group_by=None*)
Build a query to derive an alert rule from its children.

lib_alert_tree.models.**severity_pair**(*name*, *summary_name=None*, *summary_plural=False*, *warning_children=None*, *critical_children=None*, ***kwargs*)
Generate a pair of DerivedAlerts with 'warning' and 'critical' severity.

lib_alert_tree.prometheus

Classes for storing and serializing Prometheus alert rules.

class lib_alert_tree.prometheus.**AlertRule**(*name*, *expr=None*, *duration=None*, *annotations=None*, *labels=None*, *severity=None*, *summary=None*)

A single alerting rule.

property child_id
A short representation of this alert, for use in annotations.

property child_json_path
A JSONPath filter expression for selecting this alert as a child.

This expression will be combined into a full JSONPath query for retrieving all children of a derived alert, exposed in an annotation for consumption by clients (such as UIs).

format_labels(***updates*)
Format labels (and optional updates) as a string.

labels_to_json_path_filters(***updates*)
Build JSON Path filters matching the labels.

property query
The PromQL query for selecting this alert.

serialize()
Serialize this data container into a dict.

class lib_alert_tree.prometheus.**PrometheusRule**(*name*, *namespace*, *labels=None*, *groups=None*)

A complete PrometheusRule custom resource.

serialize()
Serialize this data container into a dict.

class lib_alert_tree.prometheus.**RulesGroup**(*name*, *rules=None*)

A group of alerting rules.

serialize()
Serialize this data container into a dict.

class lib_alert_tree.prometheus.**Serializable**

Base-class for data serializable into YAML strings.

dump(*out=None*)
Dump the serialized data in YAML format.

abstract serialize()
Serialize this data container into a dict.

lib_alert_tree.kubernetes

Helpers for defining lists of alerts on usual Kubernetes objects.

lib_alert_tree.kubernetes.**daemonset_alerts**(*name*, *severity='warning'*, *namespace='default'*)
Common alerts for DaemonSets.

lib_alert_tree.kubernetes.**deployment_alerts**(*name*, *severity='warning'*, *namespace='default'*)
Common alerts for Deployments.

lib_alert_tree.kubernetes.**job_alerts**(*name*, *severity='warning'*, *namespace='default'*)
Common alerts for Jobs.

lib_alert_tree.kubernetes.**pod_alerts**(*name*, *severity='warning'*, *namespace='default'*)
Common alerts for Pods.

lib_alert_tree.kubernetes.**statefulset_alerts**(*name*, *severity='warning'*, *namespace='default'*)
Common alerts for StatefulSets.

lib_alert_tree.cli

Generate a Click command-line interface from alert trees.

lib_alert_tree.cli.**generate_cli**(*roots*, *prometheus_rule_labels=None*)
Generate a CLI from a dict of root alerts (keys are used for root selection).

CHAPTER

FOUR

GLOSSARY

Alertmanager The Alertmanager is a service for handling alerts sent by client applications, such as *Prometheus*.

See also the official Prometheus documentation for Alertmanager.

API Server

`kube-apiserver` The Kubernetes API Server validates and configures data for the Kubernetes objects that make up a cluster, such as *Nodes* or *Pods*.

See also the official Kubernetes documentation for kube-apiserver.

Bootstrap

Bootstrap node The Bootstrap node is the first machine on which MetalK8s is installed, and from where the cluster will be deployed to other machines. It also serves as the entrypoint for upgrades of the cluster.

ConfigMap A ConfigMap is a Kubernetes object that allows one to store general configuration information such as environment variables in a key-value pair format. ConfigMaps can only be applied to namespaces and once created, they can be updated automatically without the need of restarting containers that depend on it.

See also the official Kubernetes documentation for ConfigMap.

Controller Manager

`kube-controller-manager` The Kubernetes controller manager embeds the core control loops shipped with Kubernetes, which role is to watch the shared state from *API Server* and make changes to move the current state towards the desired state.

See also the official Kubernetes documentation for kube-controller-manager.

`etcd` `etcd` is a distributed data store, which is used in particular for the persistent storage of *API Server*.

For more information, see etcd.io.

Environment An Environment is a namespace or group of namespaces with specific labels containing one or more Solutions. It allows to run multiple instances of a Solution on the same cluster, without collision between them.

Grafana Grafana is a service for analysing and visualizing metrics scraped by Prometheus.

For more information, see Grafana.

Kubeconfig A configuration file for *kubectl*, which includes authentication through embedded certificates.

See also the official Kubernetes documentation for kubeconfig.

Kubelet The kubelet is the primary "node agent" that runs on each cluster node.

See also the official Kubernetes documentation for kubelet.

Kube-state-metrics The kube-state-metrics service listens to the Kubernetes API server and generates metrics about the state of the objects.

See also the official Kubernetes documentation for kube-state-metrics.

Loki Loki is a log aggregation system designed to be cost-effective, only indexing metadata (labels).

For more details, see Loki documentation.

memberlist memberlist is a Go library that manages cluster membership and member failure detection using a gossip based protocol. memberlist is eventually consistent but converges quickly on average.

Namespace A Namespace is a Kubernetes abstraction to support multiple virtual clusters backed by the same physical cluster, providing a scope for resource names.

See also the official Kubernetes documentation for namespaces.

Node A Node is a Kubernetes worker machine - either virtual or physical. A Node contains the services required to run *Pods*.

See also the official Kubernetes documentation for Nodes.

Node manifest The YAML file describing a *Node*.

See also the official Kubernetes documentation for Nodes management.

Operator A Kubernetes operator is an application-specific controller that extends the functionality of the Kubernetes API to create, configure, and manage instances of complex applications.

See also the official Kubernetes documentation for Operator.

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.

Prometheus Node-exporter The Prometheus node-exporter is an exporter for exposing hardware and OS metrics read from the Linux Kernel. Users can typically obtain the following metrics; cpu, memory, filesystem for each Kubernetes node.

or more details, see prometheus node-exporter.

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.

Secret Kubernetes Secrets let you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys.

See also the official Kubernetes documentation for Secrets.

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`.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

l

lib_alert_tree, 237
lib_alert_tree.cli, 240
lib_alert_tree.kubernetes, 239
lib_alert_tree.models, 237
lib_alert_tree.prometheus, 239

INDEX

Symbols

__version__ (*in module lib_alert_tree*), 237

A

alert_rule (*lib_alert_tree.models.BaseAlert property*), 237
alert_rule (*lib_alert_tree.models.DerivedAlert property*), 238
alert_rule (*lib_alert_tree.models.ExistingAlert property*), 238
Alertmanager, **241**
AlertRule (*class in lib_alert_tree.prometheus*), 239
API Server, **241**

B

BaseAlert (*class in lib_alert_tree.models*), 237
Bootstrap, **241**
Bootstrap node, **241**
build_json_path() (*lib_alert_tree.models.Relationship static method*), 238
build_query() (*lib_alert_tree.models.Relationship method*), 238
build_rules_group() (*lib_alert_tree.models.DerivedAlert method*), 238
build_tree() (*lib_alert_tree.models.DerivedAlert method*), 238

C

child_id (*lib_alert_tree.prometheus.AlertRule property*), 239
child_json_path (*lib_alert_tree.prometheus.AlertRule property*), 239
ConfigMap, **241**
Controller Manager, **241**
create_node() (*lib_alert_tree.models.BaseAlert method*), 237
critical() (*lib_alert_tree.models.BaseAlert class method*), 237

D

daemonset_alerts() (*in module lib_alert_tree.kubernetes*), 239
deployment_alerts() (*in module lib_alert_tree.kubernetes*), 239
DerivedAlert (*class in lib_alert_tree.models*), 238
dump() (*lib_alert_tree.prometheus.Serializable method*), 239

E

Environment, **241**
etcd, **241**
ExistingAlert (*class in lib_alert_tree.models*), 238

F

format_labels() (*lib_alert_tree.prometheus.AlertRule method*), 239

G

generate_cli() (*in module lib_alert_tree.cli*), 240
Grafana, **241**

J

job_alerts() (*in module lib_alert_tree.kubernetes*), 239
json_path (*lib_alert_tree.models.BaseAlert property*), 238

K

kube-apiserver, **241**
kube-controller-manager, **241**
kube-scheduler, **242**
Kube-state-metrics, **241**
Kubeconfig, **241**
kubectl, **243**
Kubelet, **241**

L

labels_to_json_path_filters() (*lib_alert_tree.prometheus.AlertRule method*), 239
lib_alert_tree
    module, 237
lib_alert_tree.cli

module, 240
lib_alert_tree.kubernetes
module, 239
lib_alert_tree.models
module, 237
lib_alert_tree.prometheus
module, 239
Loki, 242

M

memberlist, 242
module
lib_alert_tree, 237
lib_alert_tree.cli, 240
lib_alert_tree.kubernetes, 239
lib_alert_tree.models, 237
lib_alert_tree.prometheus, 239

N

Namespace, 242
Node, 242
Node manifest, 242

O

Operator, 242

P

Pod, 242
pod_alerts() (*in module lib_alert_tree.kubernetes*), 239
pretty_str (*lib_alert_tree.models.BaseAlert property*), 238
Prometheus, 242
Prometheus Node-exporter, 242
PrometheusRule (*class in lib_alert_tree.prometheus*), 239

Q

query (*lib_alert_tree.models.BaseAlert property*), 238
query (*lib_alert_tree.prometheus.AlertRule property*), 239

R

Relationship (*class in lib_alert_tree.models*), 238
RulesGroup (*class in lib_alert_tree.prometheus*), 239

S

Salt Master, 242
Salt Minion, 242
SaltAPI, 242
Scheduler, 242
Secret, 243
Serializable (*class in lib_alert_tree.prometheus*), 239
serialize() (*lib_alert_tree.prometheus.AlertRule method*), 239
serialize() (*lib_alert_tree.prometheus.PrometheusRule method*), 239
serialize() (*lib_alert_tree.prometheus.RulesGroup method*), 239
serialize() (*lib_alert_tree.prometheus.Serializable method*), 239
Service, 243
severity_pair() (*in module lib_alert_tree.models*), 238
statefulset_alerts() (*in module lib_alert_tree.kubernetes*), 240

T

Taint, 243
Toleration, 243

W

warning() (*lib_alert_tree.models.BaseAlert class method*), 238