KaaS Reference Architecture

version beta
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copyright notice</td>
<td>1</td>
</tr>
<tr>
<td>Preface</td>
<td>2</td>
</tr>
<tr>
<td>Intended audience</td>
<td>2</td>
</tr>
<tr>
<td>Documentation history</td>
<td>2</td>
</tr>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>Mirantis KaaS overview</td>
<td>4</td>
</tr>
<tr>
<td><strong>KaaS components stack</strong></td>
<td>7</td>
</tr>
<tr>
<td>KaaS provider</td>
<td>7</td>
</tr>
<tr>
<td>KaaS release controller</td>
<td>9</td>
</tr>
<tr>
<td>KaaS web UI</td>
<td>10</td>
</tr>
<tr>
<td>KaaS bare metal</td>
<td>11</td>
</tr>
<tr>
<td>IP Address Management</td>
<td>13</td>
</tr>
<tr>
<td>KaaS limitations on bare metal</td>
<td>13</td>
</tr>
<tr>
<td>Kubernetes lifecycle management</td>
<td>13</td>
</tr>
<tr>
<td>LCM custom resources</td>
<td>14</td>
</tr>
<tr>
<td>LCM controller</td>
<td>15</td>
</tr>
<tr>
<td>LCM agent</td>
<td>16</td>
</tr>
<tr>
<td>Helm controller</td>
<td>16</td>
</tr>
<tr>
<td>Identity and access management</td>
<td>17</td>
</tr>
<tr>
<td>IAM API and CLI</td>
<td>17</td>
</tr>
<tr>
<td>External Identity Provider integration</td>
<td>18</td>
</tr>
<tr>
<td>Authentication and authorization</td>
<td>19</td>
</tr>
<tr>
<td>Implementation flow</td>
<td>19</td>
</tr>
<tr>
<td>Kubernetes CLI authentication flow</td>
<td>19</td>
</tr>
<tr>
<td><strong>Storage</strong></td>
<td>20</td>
</tr>
<tr>
<td>Overview</td>
<td>20</td>
</tr>
<tr>
<td>Limitations</td>
<td>22</td>
</tr>
<tr>
<td><strong>Monitoring</strong></td>
<td>23</td>
</tr>
<tr>
<td>Deployment architecture</td>
<td>23</td>
</tr>
<tr>
<td>Authentication flow</td>
<td>25</td>
</tr>
<tr>
<td>Supported features</td>
<td>27</td>
</tr>
</tbody>
</table>
Monitored components  

Requirements  

Baremetal-based KaaS cluster  
Reference hardware configuration  
System requirements for the seed node  
Host networking  
Cluster networking  
Network fabric  
Host storage  
OpenStack-based KaaS cluster  
AWS-based KaaS cluster
Copyright notice

2020 Mirantis, Inc. All rights reserved.

This product is protected by U.S. and international copyright and intellectual property laws. No part of this publication may be reproduced in any written, electronic, recording, or photocopying form without written permission of Mirantis, Inc.

Mirantis, Inc. reserves the right to modify the content of this document at any time without prior notice. Functionality described in the document may not be available at the moment. The document contains the latest information at the time of publication.

Mirantis, Inc. and the Mirantis Logo are trademarks of Mirantis, Inc. and/or its affiliates in the United States an other countries. Third party trademarks, service marks, and names mentioned in this document are the properties of their respective owners.
Preface
This documentation provides information on how to use Mirantis products to deploy cloud environments. The information is for reference purposes and is subject to change.

Intended audience
This documentation assumes that the reader is familiar with network and cloud concepts and is intended for the following users:

- Infrastructure Operator
  - Is member of the IT operations team
  - Has working knowledge of Linux, virtualization, Kubernetes API and CLI, and OpenStack to support the application development team
  - Accesses Mirantis KaaS and Kubernetes through a local machine or web UI
  - Provides verified artifacts through a central repository to the Tenant DevOps engineers
- Tenant DevOps engineer
  - Is member of the application development team and reports to line-of-business (LOB)
  - Has working knowledge of Linux, virtualization, Kubernetes API and CLI to support application owners
  - Accesses Mirantis KaaS and Kubernetes through a local machine or web UI
  - Consumes artifacts from a central repository approved by the Infrastructure Operator

Documentation history
The following table lists the released revisions of this documentation:

<table>
<thead>
<tr>
<th>Revision date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 19, 2019</td>
<td>Mirantis KaaS Beta</td>
</tr>
</tbody>
</table>
Introduction

Mirantis Kubernetes-as-a-Service (KaaS) can be used to orchestrate the Kubernetes clusters and addresses key challenges with running Kubernetes on-premises with pure open-source software, including:

- Distribution-agnostic Kubernetes cluster management capabilities utilizing Cluster API and Kubespray, with self-service API and web-based UI
- Controlling and delegating access to Kubernetes clusters and namespaces using existing Identity Providers with IAM integration based on Keycloak
- Backend-agnostic load balancing and storage capabilities for Kubernetes through integration with OpenStack Octavia and Cinder APIs
- Deployment automation of the Kubernetes Dashboard
Mirantis KaaS overview

Mirantis Kubernetes-as-a-Service (KaaS) is a set of microservices that are deployed using Helm charts and run in a Kubernetes cluster. Mirantis KaaS is based on the Kubernetes Cluster API community initiative.

The following diagram illustrates the Mirantis KaaS overview:

By default, KaaS provides a caching proxy that allows downloading artifacts from the Mirantis sources. Proxy works in the forward mode, it is accessed directly by replacing the target resource host name with the proxy host name. KaaS performs all steps that require the proxy usage automatically, making proxying transparent to the application owners.

All artifacts used by Kubernetes and workloads are stored on the Mirantis KaaS content delivery network (CDN):

- mirror.mirantis.com (Debian packages including the Ubuntu mirrors)
- binary.mirantis.com (Helm charts and binary artifacts)
- mirantis.azurecr.io (Docker image registry)

All artifacts are consumed through proxy. The proxy address is dynamic and set by KaaS, so the base URL is also dynamic and configurable on the workloads side. The KaaS child clusters do not require Internet access.

All Mirantis KaaS components are deployed in the Kubernetes clusters. All KaaS APIs are implemented using the Kubernetes Custom Resource Definition (CRD) that represents custom objects stored in Kubernetes and allows you to expand Kubernetes API.

The Mirantis KaaS logic is implemented using controllers. A controller handles the changes in custom resources defined in the controller CRD. A custom resource consists of a spec that describes the desired state of a resource provided by a user. During every change, a controller reconciles the external state of a custom resource with the user parameters and stores this external state in the status subresource of its custom resource.

The types of the Mirantis KaaS clusters include:

**Bootstrap cluster**
• Runs the bootstrap process on a seed node. For an OpenStack-based or AWS-based KaaS cluster, it can be an operator desktop computer, for a bare metal KaaS cluster, this is the first temporary data center node.
• Requires access to a provider back end, OpenStack, AWS, or bare metal.
• Contains minimum set of services to deploy the management and regional clusters.
• Is destroyed completely after a successful bootstrap.

**Management and regional clusters**

• Management cluster:
  • Runs all public APIs and services including the web UIs of a KaaS cluster.
  • Does not require access to any provider back end.
• Regional cluster:
  • Is combined with management cluster by default.
  • Runs the provider-specific services and internal API including LCMMachine and LCMCluster. Also, it runs an LCM controller for orchestrating child clusters and other controllers for handling different resources.
  • Requires two-way access to a provider back end. The provider connects to a back end to spawn a child cluster nodes, and the agent running on the nodes accesses the regional cluster to obtain the deployment information.
  • Requires access to a management cluster to obtain user parameters.

**Child cluster**

• A Kubernetes cluster that an end user creates using KaaS.
  • Requires access to a regional cluster. Each node of a child cluster runs an LCM agent that connects to the LCM machine of the regional cluster to obtain the deployment details.

The following diagram illustrates the distribution of services between each type of the Mirantis KaaS clusters:
KaaS components stack

Mirantis KaaS comprises a strict set of components of specific versions to be deployed on a Mirantis KaaS cluster.

KaaS provider

The KaaS provider is the central component of Mirantis Kaas that provisions a Kubernetes child cluster node and runs the LCM agent on this node. It runs in a regional cluster and requires connection to a provider back end.

The KaaS provider interacts with the following types of public API objects:

<table>
<thead>
<tr>
<th>Public API object name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KaaS release object</td>
<td>Contains the following information about a KaaS management and child cluster:</td>
</tr>
<tr>
<td></td>
<td>• Version of the supported Cluster release for a KaaS management cluster</td>
</tr>
<tr>
<td></td>
<td>• List of supported Cluster releases for the KaaS child clusters and supported upgrade path</td>
</tr>
<tr>
<td></td>
<td>• Description of Helm charts that are installed on the KaaS management clusters depending on the selected provider</td>
</tr>
</tbody>
</table>
| Cluster release object | • Provides a specific version of a KaaS management and child cluster. Any Cluster release object, as well as a KaaS release object never changes, only new releases can be added. Any change leads to a new release of a cluster.  
• Contains references to all components and their versions that are used to deploy the KaaS management and child clusters:  
  • Kubernetes LCM components:  
    • LCM agent  
    • Ansible playbooks  
    • Scripts  
    • Description of steps to execute during a Kubernetes cluster deployment and upgrade  
    • Helm controller image references  
  • Supported Helm charts description:  
    • Helm chart name and version  
    • Helm release name  
    • Helm values |
| --- | --- |
| Cluster object | • References the Credentials, KaaS, and Cluster release objects.  
• Is tied to a specific KaaS region and provider.  
• Represents all cluster-level resources. For example, for the OpenStack-based clusters, it represents networks, load balancer for the Kubernetes API, and so on. It uses data from the Credentials object to create these resources and data from the KaaS and Cluster release objects to ensure that all lower-level cluster objects are created. |
| Machine object | • References the Cluster object.  
• Represents one node of a child cluster, for example, an OpenStack VM, and contains all data to provision it. |
| Credentials object | • Contains all information necessary to connect to a provider back end. For example, clouds.yaml for the OpenStack-based clusters.  
• Is tied to a specific KaaS region and provider. |
| PublicKey object | Is provided to every machine to obtain an SSH access. |
The following diagram illustrates the KaaS provider data flow:

![Diagram of KaaS provider data flow]

The KaaS provider performs the following operations in Mirantis KaaS:

- Consumes the below types of data from a management cluster:
  - Credentials to connect to a provider back end
  - Deployment instructions from the KaaS and Cluster release objects
  - The cluster-level parameters from the Cluster objects
  - The machine-level parameters from the Machine objects
- Prepares data for all Mirantis KaaS components:
  - Creates the LCMCluster and LCMMachine custom resources for LCM controller and LCM agent. The LCMMachine custom resources are created empty to be later handled by the LCM controller.
  - Creates the the HelmBundle custom resources for the Helm controller using data from the KaaS and Cluster release objects.
  - Creates service accounts for these custom resources.
  - Creates a scope in Identity and access management (IAM) for a user access to a child cluster.
  - Provisions nodes for a child cluster using the cloud-init script that downloads and runs the LCM agent.

**KaaS release controller**

In Mirantis KaaS, the KaaS release controller is responsible for the following functionality:
• Monitor and control the KaaSRelease and ClusterRelease objects present in a KaaS management cluster. If any release object is used in a cluster, the KaaS release controller prevents the deletion of such an object.

• Sync the KaaSRelease and ClusterRelease objects published at https://binary.mirantis.com/releases/ with an existing KaaS management cluster.

• Trigger the KaaS auto-upgrade procedure if a new KaaSRelease object is found:

  1. Search for the KaaS child clusters with old Cluster releases that are not supported by a new KaaS release. If any are detected, abort the auto-upgrade and display a corresponding note about an old Cluster release in the KaaS web UI for the KaaS child clusters. In this case, a user must update all KaaS child clusters using the KaaS web UI. Once all child clusters are upgraded to the Cluster releases supported by a new KaaS release, the KaaS auto-upgrade is retriggereed by the KaaS release controller.

     
     
     Note
     Each KaaS release supports at least two versions of a Cluster release for the user to have enough time to update a KaaS child cluster to a Cluster release supported by a newer version of a KaaS release.

  2. Trigger the KaaS release upgrade of all KaaS components in a KaaS management cluster. The upgrade itself is processed by the KaaS provider.

  3. Trigger the Cluster release upgrade of a KaaS management cluster to the Cluster release version that is indicated in the upgraded KaaS release version.

     Once a KaaS management cluster is upgraded, an option to update a KaaS child cluster becomes available in the KaaS web UI. During a KaaS child cluster update, all cluster components including Kubernetes are automatically upgraded to newer versions if available.

KaaS web UI

The KaaS web UI is mainly designed to create and update the KaaS child clusters as well as add or remove machines to or from an existing child cluster.

You can also use the KaaS web UI to obtain the KaaS management cluster details including endpoints, release version, and so on. The KaaS management cluster update occurs automatically with a new release change log available through the KaaS web UI.

The KaaS web UI is a JavaScript application that is based on the React framework. The KaaS web UI designed to work on a client side only. Therefore, it does not require a special back end. It interacts with the Kubernetes and Keycloak APIs directly. The KaaS web UI uses a Keycloak token to interact with a Kubernetes cluster and download kubeconfig for the KaaS management and child clusters.

The KaaS web UI uses NGINX that runs on a management cluster and handles the KaaS web UI static files. NGINX proxies the Kubernetes and Keycloak APIs for the KaaS web UI.
KaaS bare metal

The bare metal service provides for the discovery, deployment, and management of bare metal hosts.

The bare metal management in Mirantis KaaS is implemented as a set of modular microservices. Each microservice implements a certain requirement or function within the bare metal management system.

The bare metal management solution for Mirantis KaaS includes the following components:

<table>
<thead>
<tr>
<th>KaaS bare metal components</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Component</strong></td>
</tr>
<tr>
<td>OpenStack Ironic</td>
</tr>
<tr>
<td>OpenStack Ironic Inspector</td>
</tr>
<tr>
<td>Bare Metal Operator</td>
</tr>
<tr>
<td>cluster-api-provider-baremetal</td>
</tr>
<tr>
<td>OpenStack Ironic Python Agent (IPA)</td>
</tr>
<tr>
<td>LCM agent</td>
</tr>
<tr>
<td>Ceph</td>
</tr>
<tr>
<td>MetalLB</td>
</tr>
<tr>
<td>NGINX</td>
</tr>
<tr>
<td>Keepalived</td>
</tr>
<tr>
<td>IPAM</td>
</tr>
</tbody>
</table>

The diagram below summarizes the following components and resource kinds:

- Metal³-based bare metal management in KaaS (white)
- Internal APIs (yellow)
• External dependency components (blue)

See also
KaaS limitations on bare metal
**IP Address Management**

Starting from the KaaS release 1.3.0, KaaS on bare metal uses the IP Address Management (IPAM) to keep track of the network addresses allocated to bare metal hosts. This is necessary to avoid IP address conflicts and expiration of address leases to machines through DHCP.

The IPAM functions are provided by the kaas-ipam controller and a set of custom resources. A cluster API extension enables you to define the addresses and associate them with hosts. The addresses are then configured by the Ironic provisioning system using the cloud-init tool.

The kaas-ipam controller provides the following functionality:

- Link the IPAM objects with the cluster API objects, such as BareMetalHost or Machine through the intermediate IpamHost objects.
- Starting from the KaaS release 1.5.0, handle the IP pools and addresses as Kubernetes custom objects defined by CRDs. Previously, this function was provided by the inwinstack-ipam controller that was removed in the KaaS release 1.5.0.
- Control the integration with KaaS.

KaaS IPAM uses single L3 network per management cluster, as defined in Cluster networking, to assign addresses to bare metal hosts.

**KaaS limitations on bare metal**

The KaaS bare metal management system has the following limitations in support of hardware capabilities:

- Only one network interface per node is supported
  
  MetalLB does not support multiple up and configured interfaces in the system. The number of active interfaces on any server that runs MetalLB pod is limited to 1 (one).

- Multiple storage devices are required
  
  To enable Ceph storage on the KaaS management cluster nodes, multiple storage devices are required for each node. The first device is always used by the operating system. At least one additional disk per server must be configured as a Ceph OSD device. The recommended number of OSD devices per a management cluster node is two or more. KaaS supports up to 22 OSD devices per node.

- Only the UEFI boot mode is supported
  
  All bare metal hosts used to deploy KaaS management and child Kubernetes clusters must be configured to boot in the UEFI mode. The non-UEFI boot is not supported by the Bare Metal Operator component of KaaS.

**Kubernetes lifecycle management**

The Kubernetes lifecycle management (LCM) engine in Mirantis KaaS consists of the following components:

**LCM controller**

- Responsible for all LCM operations. Consumes the LCMCluster object and orchestrates actions through LCM agent.
**LCM agent**

Runs on the target host. Executes Ansible playbooks in headless mode.

**Helm controller**

Responsible for the lifecycle of the Helm charts. It is installed by LCM controller and interacts with Tiller.

The Kubernetes LCM components handle the following custom resources:

- LCMCluster
- LCMMachine
- HelmBundle

The following diagram illustrates handling of the LCM custom resources by the Kubernetes LCM components. On a child cluster, apiserver handles multiple Kubernetes objects, for example, deployments, nodes, RBAC, and so on.

![Diagram](image)

**LCM custom resources**

The Kubernetes LCM components handle the following custom resources (CRs):

- LCMMachine
- LCMCluster
- HelmBundle

**LCMMachine**

Describes a machine that is located on a cluster. It contains the machine type, control or worker, StateItems that correspond to Ansible playbooks and miscellaneous actions, for example, downloading a file or executing a shell command. LCMMachine reflects the
current state of the machine, for example, a node IP address, and each StateItem through its status. Multiple LCMMachine CRs can correspond to a single cluster.

**LCMCluster**

Describes a KaaS child cluster. In its spec, LCMCluster contains a set of StateItems for each type of LCMMachine, which describe the actions that must be performed to deploy the cluster. LCMCluster is created by the provider, using machineTypes of the Release object. The status field of LCMCluster reflects the status of the cluster, for example, the number of ready or requested nodes.

**HelmBundle**

Wrapper for Helm charts that is handled by Helm controller. HelmBundle tracks what Helm charts must be installed on a child cluster.

**LCM controller**

LCM controller runs on the regional cluster and orchestrates the LCMMachine objects according to their type and their LCMCluster object.

Once the LCMCluster and LCMMachine objects are created, LCM controller starts monitoring them to modify the spec files and update the status fields of the LCMMachine objects when required. The status field of LCMMachine is updated by LCM agent running on a Kubernetes cluster node.

Each LCMMachine has the following lifecycle states:

1. Uninitialized - the machine is not yet assigned to an LCMCluster
2. Prepare - the machine executes StateItems that correspond to the prepare phase. This phase usually involves downloading the necessary archives and installing the packages.
3. Deploy - the machine executes StateItems that correspond to the deploy phase that is becoming a Kubernetes cluster node.
4. Ready - the machine is deployed.
5. Reconfigure - the machine is being updated with a new set of control plane nodes. Once done, the machine moves to the ready state again.

The templates for StateItems are stored in the machineTypes field of an LCMCluster object, with separate lists for the Kubernetes control plane and worker nodes. Each StateItem has the execution phase field:

1. The prepare phase is executed for all machines for which it was not executed yet. This phase comprises downloading the files necessary for a cluster deployment, installing the required packages, and so on.
2. During the deploy phase, a node is added to the Kubernetes cluster. LCM controller applies the deploy phase to the nodes in the following order:
   1. First control plane node is deployed.
   2. The remaining control plane nodes are deployed one by one and the worker nodes are deployed in batches (by default, up to 50 worker nodes at the same time). After at least one control plane and one worker node are in the ready state, helm-controller is installed on the Kubernetes cluster.
3. During the reconfigure phase, the information about the nodes that comprise the control plane HA is propagated through the cluster.

LCM controller deploys and upgrades a Kubernetes cluster by setting StateItems of LCMMachine objects following the corresponding StateItems phases described above. The Kubernetes cluster upgrade process follows the same logic that is used for a new deployment, that is applying a new set of StateItems to the LCMMachines after updating the LCMCluster object. But during the upgrade, the following additional actions are performed:

- If the kubeadm bootstrap token used for a new cluster deployment expires, it is updated before LCM controller joins a new node to the existing Kubernetes cluster.
- If the existing worker node is being upgraded, LCM controller performs draining and cordonning on this node honoring the Pod Disruption Budgets. This operation prevents unexpected disruptions of the workloads.
- LCM controller verifies that the required version of helm-controller is installed.

LCM agent

LCM agent handles a single machine that belongs to the Kubernetes cluster. It runs on the machine operating system but communicates with apiserver of the regional cluster. LCM agent is deployed as a systemd unit using cloud-init. LCM agent has a built-in self-upgrade mechanism.

LCM agent monitors the spec of a particular LCMMachine object to reconcile the machine state with the object StateItems and update the LCMMachine status accordingly. The actions that LCM agent performs while handling the StateItems are as follows:

- Download configuration files
- Run shell commands
- Run Ansible playbooks in headless mode

LCM agent provides the IP address and hostname of the machine for the LCMMachine status parameter.

Helm controller

Helm controller is used by Mirantis KaaS to manage the Kubernetes cluster core addons such as StackLight and the application addons such as the OpenStack components.

Helm controller runs in the same pod as the Tiller process. The Tiller gRPC endpoint is not accessible outside the pod. The pod is created using StatefulSet inside a tenant cluster by LCM controller once the cluster contains at least one Kubernetes control plane and worker node.

The Helm release information is stored in the Release object that is used by the Kubernetes cluster actuator. The Cluster API provider actuator uses the information from the Release objects together with the Cluster API Cluster spec. In the Cluster spec, the operator can specify the Helm release name and charts to use. By combining the information from the Cluster providerSpec parameter and its Release object, the cluster actuator generates the LCMCluster objects. These objects are further handled by LCM controller and the HelmBundle object handled by Helm controller. HelmBundle must have the same name as the LCMCluster object for the tenant cluster that HelmBundle applies to.
Although a cluster actuator can only create a single HelmBundle per a tenant cluster, Helm controller can handle multiple HelmBundle objects per one Kubernetes cluster. Helm controller handles the HelmBundle objects and reconciles them with the Tiller state in its tenant cluster. However, full reconciliation against Tiller is not supported yet relying on the status data of the HelmBundle objects. Helm controller can also be used by the management cluster with corresponding HelmBundle objects created as part of the initial management cluster setup.

Identity and access management

Identity and access management (IAM) provides a central point of users and permissions management of the Mirantis KaaS cloud resources in a granular and unified manner. Also, IAM provides infrastructure for single sign-on user experience across all Mirantis KaaS web portals.

IAM for Mirantis KaaS consists of the following components:

**Keycloak**

- Provides the OpenID Connect endpoint
- Integrates with an external Identity Provider (IdP), for example, existing LDAP or Google Open Authorization (OAuth)
- Stores roles mapping for users

**IAM controller**

- Provides IAM API with data about KaaS namespaces
- Handles all role-based access control (RBAC) components in Kubernetes API

**IAM API**

Provides an abstraction API for creating user scopes and roles

**IAM API and CLI**

Mirantis IAM exposes the versioned and backward compatible Google remote procedure call (gRPC) protocol API to interact with IAM CLI.

IAM API is designed as a user-facing functionality. For this reason, it operates in the context of user authentication and authorization.

In IAM API, an operator can use the following entities:

- Grants - to grant or revoke user access
- Scopes - to describe user roles
- Users - to provide user account information

KaaS UI interacts with IAM API on behalf of the user. However, the user can directly work with IAM API using IAM CLI. IAM CLI uses the OpenID Connect (OIDC) endpoint to obtain the OIDC token for authentication in IAM API and enable you to perform different API operations.

The following diagram illustrates the interaction between IAM API and CLI:
External Identity Provider integration

To be consistent and keep the integrity of a user database and user permissions, in Mirantis KaaS, IAM stores the user identity information internally. However in real deployments, the identity provider usually already exists.

Out of the box, in Mirantis KaaS, IAM supports integration with LDAP and Google Open Authorization (OAuth). If LDAP is configured as an external identity provider, IAM performs one-way synchronization doing mapping attributes according to configuration.

In case of the Google Open Authorization (OAuth) integration, the user is automatically registered and their credentials are stored in the internal database according to the user template configuration. The Google OAuth registration workflow is as follows:

1. The user requests a KaaS web UI resource.
2. The user is redirected to the IAM login page and logs in using the Log in with Google account option.
3. IAM creates a new user with the default access rights that are defined in the user template configuration.
4. The user can access the KaaS web UI resource.

The following diagram illustrates the external IdP integration to IAM:
You can configure simultaneous integration with both external IdPs with the user identity matching feature enabled.

Authentication and authorization

In Mirantis KaaS, IAM uses the OpenID Connect (OIDC) protocol for handling authentication.

Implementation flow

In Mirantis KaaS, IAM performs as an OpenID Connect (OIDC) provider, it issues a token and exposes discovery endpoints.

The credentials can be handled by IAM itself or delegated to an external identity provider (IdP).

The issued token is sufficient to perform operations across KaaS according to the scope and role defined in it. Mirantis recommends using asymmetric cryptography for token signing (RS256) to minimize the dependency between IAM and managed components.

The authorization implementation is out of the scope of IAM in Mirantis KaaS and is delegated to the component level. IAM interacts with a Mirantis KaaS component using the OIDC token content that is processed by a component itself and required authorization is enforced. Such an approach enables you to have any underlying authorization that is not dependent on IAM and still to provide a unified user experience across all Mirantis KaaS components.

See also

External Identity Provider integration

Kubernetes CLI authentication flow

The following diagram illustrates the Kubernetes CLI authentication flow. The authentication flow for Helm and other Kubernetes-oriented CLI utilities is identical to the Kubernetes CLI flow, but JSON Web Tokens (JWT) must be pre-provisioned.
Storage
The baremetal-based Mirantis KaaS uses Ceph as a distributed storage system for file, block, and object storage. This section provides an overview of a Ceph cluster deployed by Mirantis KaaS.

Overview
Mirantis KaaS deploys Ceph on the baremetal-based management and child clusters using Helm charts with the following components:

• Ceph controller - a Kubernetes controller that obtains the parameters from KaaS through a custom resource (CR), creates CRs for Rook, and updates its CR status based on the Ceph cluster deployment progress. It creates users, pools, and keys for OpenStack and Kubernetes and provides Ceph configurations and keys to access them. Also, Ceph controller eventually obtains the data from the OpenStack Controller for the Keystone integration and updates the RADOS Gateway services configurations to use Kubernetes for user authentication.

• Ceph operator
  • Transforms user parameters from the KaaS web UI into Rook credentials and deploys a Ceph cluster using Rook.
  • Provides integration of the Ceph cluster with Kubernetes
  • Provides data for OpenStack to integrate with the deployed Ceph cluster
• Custom resource (CR) - represents the customization of a Kubernetes installation and allows you to define the required Ceph configuration through the KaaS web UI before deployment. For example, you can define the failure domain, pools, node roles, number of Ceph components such as Ceph OSDs, and so on.

• Rook - a storage orchestrator that deploys Ceph on top of a Kubernetes cluster.

A typical Ceph cluster consists of the following components:

**Ceph Monitors**
Three or, in rare cases, five Ceph Monitors.

**Ceph Managers**
Mirantis recommends having three Ceph Managers in every cluster

**RADOS Gateway services**
Mirantis recommends having three or more RADOS Gateway services for HA.

**Ceph OSDs**
The number of OSDs may vary according to the deployment needs.

The placement of Ceph Monitors and Ceph Managers is defined in the custom resource. The placement of other Ceph components can vary because of their migration within the Ceph cluster.

The following diagram illustrates the way a Ceph cluster is deployed in Mirantis KaaS:
The following diagram illustrates the processes within a deployed Ceph cluster:

See also
- Ceph documentation
- Rook documentation

Limitations
A Ceph cluster configuration in Mirantis KaaS includes but is not limited to the following limitations:

- Only one Ceph controller per a Kubernetes cluster and only one Ceph cluster per Ceph controller are supported.
- Only one CRUSH tree per cluster. The separation of devices per Ceph pool is supported through device classes with only one pool of each type for a device class.
- All disks of the same type must have the same size.
- All disks of the same class must have the same settings, such as the number of OSDs per device, encryption, and so on.
- All partitions of a disk must have the same size and the same role.
- Only the following types of CRUSH buckets are supported: root, room, rack, shelve, node, OSD.
- A user cannot configure each pool separately but can provide a common set of settings (relative expected data size and replication settings) for a set of pools for the same tasks (RADOS Gateway, Cinder, Nova, Glance, and so on).
- RBD mirroring is not supported.
- Consuming an existing Ceph cluster is not supported.
- CephFS is not supported.
- Only IPv4 is supported.
- If two or more OSDs are located on the same device, there must be no dedicated WAL or DB for this class.
- Only a full collocation or dedicated WAL and DB configurations are supported.
- All CRUSH rules must have the same failure_domain.
- Only the following osd_classes are supported hdd, hdd_large, ssd, ssd_large, and nvme.

**Monitoring**

Mirantis KaaS uses StackLight, the logging, monitoring, and alerting solution that provides a single pane of glass for cloud maintenance and day-to-day operations as well as offers critical insights into cloud health including operational information about the components deployed in Kubernetes clusters. StackLight is based on Prometheus, an open-source monitoring solution and a time series database.

**Deployment architecture**

Mirantis KaaS deploys the StackLight stack as a release of a Helm chart that contains the helm-controller and helmbundles.lcm.mirantis.com (HelmBundle) custom resources. The StackLight HelmBundle consists of a set of Helm charts with the StackLight components that include:

- StackLight components overview
<table>
<thead>
<tr>
<th><strong>StackLight component</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Alerta</td>
<td>Receives, consolidates, and deduplicates the alerts sent by Alertmanager and visually represents them through a simple web UI. Using Alerta, you can view the most recent or watched alerts, group, and filter alerts.</td>
</tr>
<tr>
<td>Alertmanager</td>
<td>Handles the alerts sent by client applications such as Prometheus, deduplicates, groups, and routes alerts to receiver integrations, as well as performs silencing and inhibition of alerts.</td>
</tr>
<tr>
<td>Elasticsearch</td>
<td>Stores logs and notifications.</td>
</tr>
<tr>
<td>Elasticsearch curator</td>
<td>Maintains the data (indexes) in Elasticsearch by performing such operations as creating, closing, or opening an index as well as deleting a snapshot. Also, manages the data retention policy in Elasticsearch.</td>
</tr>
<tr>
<td>Elasticsearch exporter</td>
<td>The Prometheus exporter that gathers internal Elasticsearch metrics.</td>
</tr>
<tr>
<td>Fluentd-elasticsearch</td>
<td>Collects logs, sends them to Elasticsearch, generates metrics based on analysis of incoming log entries, and exposes these metrics to Prometheus.</td>
</tr>
<tr>
<td>Grafana</td>
<td>Builds and visually represents metric graphs based on time series databases. Grafana supports querying of Prometheus using the PromQL language.</td>
</tr>
<tr>
<td>Kibana</td>
<td>Provides real-time visualization of the data stored in Elasticsearch and enables you to detect issues.</td>
</tr>
<tr>
<td>Metricbeat</td>
<td>Collects Kubernetes events and sends them to Elasticsearch for storage.</td>
</tr>
<tr>
<td>Available since KaaS 1.3.0</td>
<td></td>
</tr>
<tr>
<td>Prometheus</td>
<td>Gathers metrics. Automatically discovers and monitors the endpoints.</td>
</tr>
<tr>
<td>Prometheus-es-exporter</td>
<td>Presents the Elasticsearch data as Prometheus metrics by periodically sending configured queries to the Elasticsearch cluster and exposing the results to a scrapable HTTP endpoint like other Prometheus targets.</td>
</tr>
<tr>
<td>Prometheus node exporter</td>
<td>Gathers hardware and operating system metrics exposed by kernel.</td>
</tr>
<tr>
<td>Available since KaaS 1.3.0</td>
<td></td>
</tr>
<tr>
<td>Prometheus Relay</td>
<td>Adds a proxy layer to Prometheus to merge the results from underlay Prometheus servers to prevent gaps in case some data is missing on some servers. Is available only in the HA StackLight mode.</td>
</tr>
</tbody>
</table>
| Pushgateway | Enables ephemeral and batch jobs to expose their metrics to Prometheu

Since these jobs may not exist long enough to be scraped, they can instead push their metrics to Pushgateway, which then exposes these metrics to Prometheus. Pushgateway is not an aggregator or a distributed counter but rather a metrics cache. The pushed metrics are exactly the same as scraped from a permanently running program. |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Salesforce notifier</td>
<td>Enables creating Salesforce cases from the Alertmanager notifications.</td>
</tr>
<tr>
<td>Telegraf</td>
<td>Collects metrics from the system. Telegraf is plugin-driven and has the concept of two distinct set of plugins: input plugins collect metrics from the system, services, or third-party APIs; output plugins write and expose metrics to various destinations. The Telegraf agents used in KaaS include:</td>
</tr>
<tr>
<td>• telegraf-ds-smart monitors SMART disks, and runs on both KaaS management and child clusters.</td>
<td></td>
</tr>
<tr>
<td>• telegraf-ironic monitors Ironic on the baremetal-based KaaS management clusters. The ironic input plugin collects and processes data from Ironic HTTP API, while the http_response input plugin checks Ironic HTTP API availability. As an output plugin, to expose collected data as Prometheus target, Telegraf uses prometheus.</td>
<td></td>
</tr>
</tbody>
</table>

Every Helm chart contains a default values.yml file. These default values are partially overridden by custom values defined in the StackLight Helm chart.

Before deploying a KaaS management or child cluster, you can select the HA or non-HA StackLight architecture type. These architecture types have the following differences:

- HA StackLight mode:
  - Two Prometheus instances
  - One Prometheus Relay instance
  - Three Elasticsearch instances
- Non-HA StackLight mode:
  - One Prometheus instance
  - No Prometheus Relay instance
  - One Elasticsearch instance

**Authentication flow**

StackLight provides five web UIs including Prometheus, Alertmanager, Alerta, Kibana, and Grafana. Access to StackLight web UIs is protected by Keycloak-based Identity and access management (IAM). All web UIs except Alerta are exposed to IAM through the IAM proxy middleware. The Alerta configuration provides direct integration with IAM.
The following diagram illustrates accessing the IAM-proxied StackLight web UIs, for example, Prometheus web UI:

Authentication flow for The IAM-proxied StackLight web UIs:

1. A user enters the public IP of a StackLight web UI, for example, Prometheus web UI.
2. The public IP leads to IAM proxy, deployed as a Kubernetes LoadBalancer, which protects the Prometheus web UI.
3. LoadBalancer routes the HTTP request to Kubernetes internal IAM proxy service endpoints, specified in the X-Forwarded-Proto or X-Forwarded-Host headers.
4. The Keycloak login form opens (--discovery-url in the IAM proxy, which points to Keycloak realm) and the user enters the user name and password.
5. Keycloak validates the user name and password.
6. The user obtains access to the Prometheus web UI (--upstream-url in the IAM proxy).

Note

- The discovery URL is the URL of the IAM service.
- The upstream URL is the hidden endpoint of a web UI (Prometheus web UI in the example above).

The following diagram illustrates accessing the Alerta web UI:

Authentication flow for the Alerta web UI:
1. A user enters the public IP of the Alerta web UI.
2. The public IP leads to Alerta deployed as a Kubernetes LoadBalancer type.
3. LoadBalancer routes the HTTP request to the Kubernetes internal Alerta service endpoint.
4. The Keycloak login form opens (Alerta refers to the IAM realm) and the user enters the user name and password.
5. Keycloak validates the user name and password.
6. The user obtains access to the Alerta web UI.

Supported features

Using the Kaas web UI, on the pre-deployment stage of a child cluster, you can view, enable or disable, or tune the following StackLight features available:

- StackLight HA mode.
- Database retention size and time for Prometheus.
- Tunable index retention period for Elasticsearch.
- Tunable PersistentVolumeClaim (PVC) size for Prometheus and Elasticsearch set to 16 GB for Prometheus and 30 GB for Elasticsearch by default. The PVC size must be logically aligned with the retention periods or sizes for these components.
- Email and Slack receivers for the Alertmanager notifications.
- Predefined set of dashboards.
- Predefined set of alerts and capability to add new custom alerts for Prometheus in the following exemplary format:

```markdown
- alert: HighErrorRate
  expr: job:request_latency_seconds:mean5m{job="myjob"} > 0.5
  for: 10m
  labels:
    severity: page
  annotations:
    summary: High request latency
```

Monitored components

StackLight measures, analyzes, and reports in a timely manner about failures that may occur in the following Mirantis KaaS components and their sub-components, if any:

- Ceph
- Ironic (KaaS bare-metal provider)
- Kubernetes services:
  - Calico
  - etcd
• Kubernetes cluster
• Kubernetes containers
• Kubernetes deployments
• Kubernetes nodes
• Netchecker
  Available since KaaS 1.3.0
• MongoDB
• NGINX
• Node hardware and operating system
• SMART disks
• StackLight:
  • Alertmanager
  • Elasticsearch
  • Grafana
  • Prometheus
  • Prometheus Relay
  • Pushgateway
  • Salesforce notifier
• SSL certificates
Requirements

Using Mirantis KaaS, you can deploy a Kubernetes cluster on bare metal, OpenStack, or Amazon Web Services (AWS). Each provider requires corresponding resources described in this section.
Baremetal-based KaaS cluster
Reference hardware configuration

The following hardware configuration is used as a reference to deploy KaaS with bare metal Kubernetes clusters.

Reference hardware configuration for KaaS clusters on bare metal

<table>
<thead>
<tr>
<th>Server role</th>
<th># of servers</th>
<th>Server model</th>
<th>CPU model</th>
<th># of CPUs</th>
<th># of vCPUs</th>
<th>RAM, GB</th>
<th>SSD system disk, GB</th>
<th>SSD/DH D storage disk, GB</th>
<th>NIC model</th>
<th># of NICs</th>
</tr>
</thead>
<tbody>
<tr>
<td>KaaS management cluster</td>
<td>3</td>
<td>Supermicro 1U SY S-6018R-TDW</td>
<td>Intel Xeon E5-2620v4</td>
<td>1</td>
<td>16</td>
<td>96</td>
<td>1x960</td>
<td>2x1900</td>
<td>Intel X520-DA2</td>
<td>2</td>
</tr>
<tr>
<td>KaaS child cluster</td>
<td>6 (∗)</td>
<td>Supermicro 1U SY S-6018R-TDW</td>
<td>Intel Xeon E5-2620v4</td>
<td>1</td>
<td>16</td>
<td>96</td>
<td>1x960</td>
<td>2x1900</td>
<td>Intel X520-DA2</td>
<td>2</td>
</tr>
</tbody>
</table>

System requirements for the seed node

The seed node is necessary only to deploy the management cluster. When the bootstrap is complete, the bootstrap node can be redeployed and its resources can be reused for the child cluster workloads.

The minimum reference system requirements for a baremetal-based bootstrap seed node are as follows:

- Basic server on Ubuntu 18.04 with the following configuration:
  - Kernel version 4.15.0-76.86 or later
  - 8 GB of RAM
  - 4 CPU
  - 10 GB of free disk space for the bootstrap cluster cache
- No DHCP or TFTP servers on any NIC networks

Three control plane nodes for HA and three storage nodes for a Ceph storage cluster.
• Routable access IPMI network for the hardware servers. For more details, see Host networking.
• Internet access for downloading of all required artifacts
Host networking

The following network roles are defined for all Mirantis KaaS clusters nodes on bare metal including the bootstrap, KaaS management, and KaaS child cluster nodes:

- **Out-of-band (OOB) network**
  Connects the Baseboard Management Controllers (BMC) of the hosts in the network to Ironic. This network or multiple networks if child clusters have their own OOB networks must be accessible from the PXE network through the IP routing.

- **Common/PXE network**
  Is a general purpose network used to remotely boot servers through the PXE protocol as well as for the Kubernetes API access and Kubernetes pods traffic. This network is shared between the KaaS management and child Kubernetes clusters.

The initially installed bootstrap node or node0 must be connected to the following networks:

- The OOB network. Ironic must have access to the IPMI/BMC of the managed bare metal hosts. Though, Ironic must not be connected to the L2 segment directly. The OOB network must be accessible through the Router 1 in the PXE network.

- The Common/PXE network. The instance of the kaas-bm running on node0 provides DHCP service on this network. This service is required for Ironic to inspect the bare metal hosts and install the operating system. The bootstrap node must be directly connected to the PXE network to ensure the L2 connectivity for DHCP. The default route for node0 must point to the Router 1 in the PXE network.

The KaaS bootstrap cluster node has the following networking configuration:

![KaaS bootstrap network diagram](image)

A KaaS management cluster node has the following networking configuration:
A KaaS child cluster node has the following network configuration:
Cluster networking

The following diagram illustrates the L3 networking schema for the final state of the KaaS bare metal deployment as described in Host networking.
Network fabric

The following diagram illustrates the physical and virtual L2 underlay networking schema for the final state of the KaaS bare metal deployment.

The network fabric reference configuration is a spine/leaf with 2 leaf ToR switches and one out-of-band (OOB) switch per rack.

Reference configuration uses the following switches for ToR and OOB:

- Cisco WS-C3560E-24TD has 24 of 1 GbE ports. Used in OOB network segment.
- Dell Force 10 S4810P has 48 of 1/10GbE ports. Used as ToR in Common/PXE network segment.

In the reference configuration, all odd interfaces from NIC0 are connected to TOR Switch 1, and all even interfaces from NIC0 are connected to TOR Switch 2. The Baseboard Management Controller (BMC) interfaces of the servers are connected to OOB Switch 1.
Host storage

The following table summarizes the common mount points and sizes for a typical bare metal host.

### Bare metal host storage

<table>
<thead>
<tr>
<th>Device</th>
<th>Partition</th>
<th>Mount point</th>
<th>Recommended size, GB</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/sda</td>
<td>sda1</td>
<td>/boot/uefi</td>
<td>550 MiB</td>
<td>The EFI partition</td>
</tr>
<tr>
<td></td>
<td>sda2</td>
<td>-</td>
<td>8 MiB</td>
<td>The RAW partition for EFI</td>
</tr>
<tr>
<td></td>
<td>sda3</td>
<td>/boot</td>
<td>1 GiB</td>
<td>The Boot partition that holds the kernel images, RAMDisk images, and bootloader configuration</td>
</tr>
<tr>
<td></td>
<td>sda4</td>
<td>-</td>
<td>100% of the remaining disk image size</td>
<td>The LVM partition</td>
</tr>
<tr>
<td>/dev/vg0/lv_root</td>
<td>/</td>
<td></td>
<td>10 GB</td>
<td>(LVM) The root partition that contains the operating system file structure and all other mount points</td>
</tr>
<tr>
<td>/dev/vg0/lv_var</td>
<td>/var</td>
<td></td>
<td>100% of the remaining free space in the LVM volume group</td>
<td>(LVM) The partition that contain files to which the system writes data during the course of its operation</td>
</tr>
</tbody>
</table>

OpenStack-based KaaS cluster

While planning the deployment of an OpenStack-based Kubernetes cluster, consider the following general requirements:

- Kubernetes on OpenStack requires the Cinder and Octavia APIs availability
- The bootstrap and the management clusters must have access to *.mirantis.com to download the release information and artifacts

The following table summarizes the minimum reference requirements for an OpenStack-based Kubernetes cluster including the resources for StackLight:

### Minimum requirements for an OpenStack-based Kubernetes cluster

<table>
<thead>
<tr>
<th>Resource</th>
<th>Bootstrap cluster</th>
<th>Management cluster</th>
<th>Child cluster</th>
<th>Details</th>
</tr>
</thead>
</table>
### # of nodes

<table>
<thead>
<tr>
<th># of nodes</th>
<th>1</th>
<th>3 (HA)</th>
<th>5</th>
</tr>
</thead>
</table>

- A bootstrap cluster requires access to the OpenStack API.
- A management cluster requires 3 nodes for the control plane HA.
- A child cluster requires 3 nodes for the control plane HA and 2 nodes for the Kubernetes workloads.

### # of vCPUs per node

<table>
<thead>
<tr>
<th># of vCPUs per node</th>
<th>2</th>
<th>6</th>
<th>6</th>
</tr>
</thead>
</table>

### RAM in GB per node

<table>
<thead>
<tr>
<th>RAM in GB per node</th>
<th>4</th>
<th>16</th>
<th>16</th>
</tr>
</thead>
</table>

### Storage in GB per node

<table>
<thead>
<tr>
<th>Storage in GB per node</th>
<th>5 (available)</th>
<th>120</th>
<th>120</th>
</tr>
</thead>
</table>

### Operating system

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Ubuntu 16.04 or 18.04</th>
<th>Ubuntu 18.04</th>
<th>Ubuntu 18.04</th>
</tr>
</thead>
</table>

For a management and child cluster, a base Ubuntu 18.04 image with the default SSH ubuntu user name must be present in Glance.

### Docker version

<table>
<thead>
<tr>
<th>Docker version</th>
<th>18.09</th>
<th>-</th>
<th>-</th>
</tr>
</thead>
</table>

For a management and child cluster, containerd is deployed by KaaS instead of Docker as a CRI.

### OpenStack version

<table>
<thead>
<tr>
<th>OpenStack version</th>
<th>-</th>
<th>Queens</th>
<th>Queens</th>
</tr>
</thead>
</table>

### Obligatory OpenStack components

<table>
<thead>
<tr>
<th>Obligatory OpenStack components</th>
<th>-</th>
<th>Octavia, Cinder</th>
<th>Octavia, Cinder</th>
</tr>
</thead>
</table>

### # of Cinder volumes

<table>
<thead>
<tr>
<th># of Cinder volumes</th>
<th>-</th>
<th>7 (total 110 GB)</th>
<th>5 (total 60 GB)</th>
</tr>
</thead>
</table>

- A management cluster requires 2 volumes for KaaS (total 50 GB) and 5 volumes for StackLight (total 60 GB)
- A child cluster requires 5 volumes for StackLight

### # of load balancers

<table>
<thead>
<tr>
<th># of load balancers</th>
<th>-</th>
<th>10</th>
<th>6</th>
</tr>
</thead>
</table>

- LBs for a management cluster: 1 for Kubernetes, 4 for KaaS, 5 for StackLight
- LBs for a child cluster: 1 for Kubernetes and 5 for StackLight
# of floating IPs | 11 | 11 | -

- FIPs for a management cluster: 1 for Kubernetes, 3 for the control plane nodes (one FIP per node), 4 for KaaS, 5 for StackLight
- FIPs for a child cluster: 1 for Kubernetes, 3 for the control plane nodes, 2 for the worker nodes, 5 for StackLight

**AWS-based KaaS cluster**

While planning the deployment of an AWS-based Kubernetes cluster, consider the requirements described below.

**Warning**

Some of the AWS features required for Mirantis KaaS may not be included into your AWS account quota. Therefore, carefully consider the AWS fees applied to your account that may increase for the Mirantis KaaS infrastructure.

**Note**

The following table summarizes the reference requirements for a 50-500 node cluster used by the Mirantis validation lab for an AWS-based Kubernetes cluster including the resources for StackLight.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Bootstrap cluster</th>
<th>Management cluster</th>
<th>Child cluster</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td># of nodes</td>
<td>1</td>
<td>3 (HA)</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• A bootstrap cluster requires access to the Mirantis CDN.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• A management cluster requires three nodes for the control plane HA.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• A child cluster requires three nodes for the control plane HA and two nodes for the KaaS workloads.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td># of vCPUs per node</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAM in GB per node</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Storage in GB per node</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operating system</td>
<td>Ubuntu 16.04 or 18.04</td>
<td>Ubuntu 18.04</td>
<td>Ubuntu 18.04</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>For a management and child cluster, a base Ubuntu 18.04 image with the default SSH ubuntu user name is required.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Docker version</td>
<td>18.09</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instance type</td>
<td>-</td>
<td>c5d.2xlarge</td>
<td>c5d.2xlarge</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>c5d.4xlarge</td>
<td>c5d.4xlarge</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>r5.4xlarge</td>
<td>r5.4xlarge</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>To prevent issues with low RAM, a child cluster requires the following types of instances:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bastion host instance type</td>
<td>-</td>
<td>t2.micro</td>
<td>t2.micro</td>
<td></td>
</tr>
<tr>
<td># of volumes</td>
<td>-</td>
<td>7 (total 110 GB)</td>
<td>5 (total 60 GB)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A management cluster requires 2 volumes for KaaS (total 50 GB) and 5 volumes for StackLight (total 60 GB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># of Elastic load balancers to be used</td>
<td>-</td>
<td>10</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elastic LBs for a management cluster: 1 for Kubernetes, 4 for KaaS, 5 for StackLight</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># of Elastic IP addresses to be used</td>
<td>-</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elastic LBs for a child cluster: 1 for Kubernetes and 5 for StackLight</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The bootstrap cluster is necessary only to deploy the management cluster. When the bootstrap is complete, this cluster can be deleted and its resources can be reused for the child cluster workloads.

The recommended instance types for different cluster sizes are as follows:

- c5d.2xlarge - <50 nodes cluster
- c5d.4xlarge - 50-500 nodes cluster
- c5d.9xlarge - 500-1000 nodes cluster