metal-stack.io

Documentation Version v0.22.1

Table of contents:

Welcome to the metal-stack docs!

metal-stack is an open source software that provides an API for provisioning and managing physical servers in the data center. To categorize this product, we use the terms *Metal-as-a-Service (MaaS)* or *bare metal cloud*.

From the perspective of a user, the metal-stack does not feel any different from working with a conventional cloud provider. Users manage their resources (machines, networks and ip addresses, etc.) by themselves, which effectively turns your data center into an elastic cloud infrastructure.

The major difference to other cloud providers is that compute power and data reside in your own data center.

Getting Started

Before starting to buy any hardware, you should try out the metal-stack on your notebook and familiarize with the software.

For this, we made the mini-lab.

The mini-lab is a fully virtual setup of metal-stack and is supposed to be run locally on a single machine. For this reason, the setup was slightly simplified in comparison to full-blown setups on real hardware. However, the lab should help to understand all ideas behind the metal-stack.

Get your hands dirty and follow the guide on how to get on with the mini-lab here.

Why Bare Metal?

Bare metal has several advantages over virtual environments and overcomes several drawbacks of virtual machines. We also listed drawbacks of the bare metal approach. Bare in mind though that it is still possible to virtualize on bare metal environments when you have your stack up and running.

Virtual Environment Drawbacks

- Spectre and Meltdown can only be mitigated with a "cluster per tenant" approach
- Missing isolation of multi-tenant change impacts
- Licensing restrictions
- Noisy-neighbors

Bare Metal Advantages

- Guaranteed and fastest possible performance (especially disk i/o)
- Reduced stack depth (Host / VM / Application vs. Host / Container)
  - Reduced attack surface
  - Lower costs, higher performance
  - No VM live-migrations
- Bigger hardware configurations possible (hypervisors have restrictions, e.g. it is not possible to assign all CPUs to a single VM)

Bare Metal Drawbacks

- Hardware defects have direct impact (should be considered by design) and can not be mitigated by live-migration as in virtual environments
- Capacity planning is more difficult (no resource overbooking possible)

Why metal-stack?

Before we started with our mission to implement the metal-stack, we decided on a couple of key characteristics and constraints that we think are unique in the domain (otherwise we would definitely have chosen an existing solution).

We hope that the following properties appeal to you as well.

On-Premise

Running on-premises gives you data sovereignty and usually a better price / performance ratio than with hyperscalers — especially the larger you grow your environment. Another benefit of running on-premises is an easier connectivity to existing company networks.

Fast Provisioning

Provisioning bare metal machines should not feel much different from virtual machines. metal-stack is capable of provisioning servers in less than a minute. The underlying network topology is based on BGP and allows announcing new routes to your host machines in a matter of seconds.

No-Ops

Part of the metal-stack runs on dedicated switches in your data center. This way, it is possible to automate server inventorization, permanently reconcile network configuration and automatically manage machine lifecycles. Manual configuration is neither required nor wanted.

Security

Our networking approach was designed for highest standards on security. Also, we enforce firewalling on dedicated tenant firewalls before users can establish connections to other networks than their private tenant network. API authentication and authorization is done with the help of OIDC.

API driven

The development of metal-stack is strictly API driven and offers self-service to end-users. This approach delivers the highest possible degree of automation, maintainability and performance.

Ready for Kubernetes

Not only does the metal-stack run smoothly on Kubernetes (K8s). The major intent of metal-stack has always been to build a scalable machine infrastructure for *Kubernetes as a Service (KaaS)*. In partnership with the open-source project Gardener, we can provision Kubernetes clusters on metal-stack at scale.

From the perspective of the Gardener, the metal-stack is just another cloud provider. The time savings compared to providing machines and Kubernetes by hand are significant. We actually want to be able to compete with offers of public cloud providers, especially regarding speed and usability.

Of course, you can use metal-stack only for machine provisioning as well and just put something else on top of your metal infrastructure.

Open Source

The metal-stack is open source and free of constraints regarding vendors and third-party products. The stack is completely built on open source products. We have a community actively working on the metal-stack, which can assist you delivering all reasonable features you are gonna need.

Flavors of metal-stack

While metal-stack itself provides access to manage resources like machines, networks and ip addresses, it does not provide any higher abstractions on top when used on its own.

As modern infrastructure and cloud native applications are designed with Kubernetes in mind, we provide two different layers on top of metal-stack to provide provisioning of clusters.

Plain

Regardless which flavor of metal-stack you use, it is always possible to manually provision machines, networks and ip addresses. This is the most basic way of using metal-stack and is very similar to how traditional bare metal infrastructures are managed.

Using plain metal-stack without additional layer was not a focus in the past. Therefore firewall and role management might be premature. These will be addressed by MEP-4 and MEP-16 in the future.

Gardener

We recommend using metal-stack with our Gardener integration, which allows to manage Kubernetes clusters at scale. This integration is battle proof, well documented, used by many organizations in production and build on top of the open-source project Gardener.

When compared to our Cluster API integration, this is more and provides a lot more features and stability. Clusters can more easily be created and managed.

Cluster API

Our Cluster API integration is a more experimental approach to provide Kubernetes clusters with metal-stack. It is based on the Cluster API project.

Resulting clusters are as minimal as possible and need to be configured manually after creation. With this approach there is no concept of service clusters. Each cluster is manually created and managed.

Client Libraries

Our public-facing APIs are built on swagger, which allows you generating API clients in all sorts of programming languages.

For the metal-api we officially support the following client libraries:

- metal-go
- metal-python

Hardware Support

In order to keep the automation and maintenance overhead small, we strongly advise against building highly heterogeneous environments with metal-stack. Having a lot of different vendors and server models in your partitions will heavily increase the time and effort for introducing metal-stack in your infrastructure. From experience we can tell that the interfaces for automating hardware provisioning are usually inconsistent between vendors and even between server models of the same vendor. Therefore, we encourage adopters to start off with only a small amount of machine types. If you want to be on the safe side, you should consider buying the hardware that we officially support.

We came up with a repository called go-hal, which includes the interface required for metal-stack to support a machine vendor. If you plan to implement support for new vendors, please check out this repository and contribute back your efforts in order to make the community benefit from extended vendor support as well.

Servers

The following server types are officially supported and verified by the metal-stack project:

Vendor	Series	Model	Board Type	Status
Supermicro	Big-Twin	SYS-2029BT-HNR	X11DPT-B	stable
Supermicro	Big-Twin	SYS-220BT-HNTR	X12DPT-B6	stable
Supermicro	SuperServer	SSG-5019D8-TR12P	X11SDV-8C-TP8F	stable
Supermicro	SuperServer	2029UZ-TN20R25M	X11DPU	stable
Supermicro	SuperServer	SYS-621C-TN12R	X13DDW-A	stable
Supermicro	Microcloud	5039MD8-H8TNR	X11SDD-8C-F	stable
Supermicro	Microcloud	SYS-531MC-H8TNR	X13SCD-F	stable
Supermicro	Microcloud	3015MR-H8TNR	H13SRD-F	stable
Lenovo	ThinkSystem	SD530		alpha
Gigabyte	OCP Open Rack line			alpha

Other server series and models might work but were not reported to us.

GPUs

The following GPU types are officially supported and verified by the metal-stack project:

Vendor	Model	Status
NVIDIA	RTX 6000	stable
NVIDIA	H100	stable

Other GPU models might work but were not reported to us. For a detailed description howto use GPU support in a kubernetes cluster please check this documentation.

Network Cards

The following network cards are officially supported and verified by the metal-stack project for usage in servers:

Vendor	Series	Model	Status
Intel	XXV710	DA2 DualPort 2×25G SFP28	stable
Intel	E810	DA2 DualPort 2×25G SFP28	stable
Intel	E810	CQDA2 DualPort 2×100G QSFP28	stable
Mellanox	ConnectX-5	MCX512A-ACAT 2×25G SFP28	stable

Switches

The following switch types are officially supported and verified by the metal-stack project:

Vendor	Series	Model	OS	Status
Edge-Core	AS4600 Series	AS4625-54T	Edgecore SONiC	stable
Edge-Core	AS4600 Series	AS4630-54TE	Edgecore SONiC	stable
Edge-Core	AS7700 Series	AS7712-32X	Cumulus 3.7.13	stable
Edge-Core	AS7700 Series	AS7726-32X	Cumulus 3.7.13	stable
Edge-Core	AS7700 Series	AS7712-32X	Edgecore SONiC	stable
Edge-Core	AS7700 Series	AS7726-32X	Edgecore SONiC	stable

Other switch series and models might work but were not reported to us.

WARNING

On our switches we run SONiC. The metal-core writes network configuration specifically implemented for this operating system. Please also consider running SONiC on your switches if you do not want to run into any issues with networking.

Our previous support for Cumulus Linux will come to an end.

Of course, contributions for supporting other switch vendors and operating systems are highly appreciated.

Portable metal-stack Setup

A minimal physical hardware setup may contain at least the following components:

WARNING

This setup dedicated to testing environments, getting to know the metal-stack software and discussing BOMs for production setups.

#	Vendor	Series	Model	Function
1x	EdgeCore	AS5500 Series	AS4630-54x (1G)	Management Switch and Management Server
2x	EdgeCore	AS5500 Series	AS4625-54x (1G)	Leaf switches
1x	Supermicro	Microcloud	3015MR-H8TNR	Usable machines
1x	Teltonika	Router	RUTXR1	Front router for internet and out-of-band access to servers and switches

This setup will yield in 8 usable machines, one of them can be configured to provide persistent CSI storage.


ProCase
metalstack
cloud

Operating Systems

Our operating system images are built on regular basis from the metal-images repository.

All images are hosted on GKE at images.metal-stack.io. Feel free to use this as a mirror for your metal-stack partitions if you want. The metal-stack developers continuously have an eye on the supported images. They are updated regularly and scanned for vulnerabilities.

Supported OS Images

The operating system images that we build are trimmed down to their bare essentials for serving as Kubernetes worker nodes. Small image sizes make machine provisioning blazingly fast.

The supported images for worker nodes currently are:

Platform	Distribution	Version
Linux	Debian	12
Linux	Debian	13
Linux	Ubuntu	24.04

The supported images for firewalls are:

Platform	Distribution	Version	Based On
Linux	Ubuntu	3	24.04

Building Your Own Images

It is fully possible to build your own operating system images and provide them through the metal-stack.

There are some conventions though that you need to follow in order to make your image installable through the metal-hammer. You should understand the machine provisioning sequence before starting to write your own images.

1. Images need to be compressed to a tarball using the lz4 compression algorithm.
2. An `md5` checksum file with the same name as the image archive needs to be provided in the download path along with the actual os image.
3. A `packages.txt` containing the packages contained in the OS image should be provided in the download path (not strictly required).
4. Consider semantic image versioning, which we use in our algorithms to select latest images (e.g. `os-major.minor.patch` ➡️ `ubuntu-19.10.20191018`)

5. Consider installing packages used by the metal-stack infrastructure:
   - FRR to enable routing-to-the-host in our network topology
   - go-lldpd to enable checking if the machine is still alive after user allocation
   - ignition for enabling users to run user-specific initialization instructions before bootup. It's pretty small in size, which is why we use it. However, you are free to use other cloud instance initialization tools if you want to.
6. You have to provide an `install.sh` script, which applies user-specific configuration in the installed image
   - This script should consume parameters from the `install.yaml` file that the metal-hammer writes to `/etc/metal/install.yaml`
   - Please check this contract between image and the metal-hammer here
7. For the time being, your image must be able to support kexec into the new operating system kernel, the `kexec` command is issued by the metal-hammer after running the `install.sh`. We do this because `kexec` is *much* faster than rebooting a machine.
8. We recommend building images from Dockerfiles as it is done in metal-images repository.

> **INFO**
>
> Building own operating system images is an advanced topic. When you have just started with metal-stack, we recommend using the public operating system images first.

Deployment Guide

We are bootstrapping the metal control plane as well as our partitions with Ansible through CI.

In order to build up your deployment, we recommend to make use of the same Ansible roles that we are using by ourselves in order to deploy the metal-stack. You can find them in the repository called metal-roles.

In order to wrap up deployment dependencies there is a special deployment base image hosted on GitHub that you can use for running the deployment. Using this Docker image eliminates a lot of moving parts in the deployment and should keep the footprints on your system fairly small and maintainable.

This document will from now on assume that you want to use our Ansible deployment roles for setting up metal-stack. We will also use the deployment base image, so you should also have Docker installed. It is in the nature of software deployments to differ from site to site, company to company, user to user. Therefore, we can only describe you the way of how the deployment works for us. It is up to you to tweak the deployment described in this document to your requirements.

WARNING

Probably you need to learn writing Ansible playbooks if you want to be able to deploy the metal-stack as presented in this documentation. However, even when starting without any knowledge about Ansible it should be possible to follow these docs. In case you need further explanations regarding Ansible please refer to docs.ansible.com.

INFO

If you do not want to use Ansible for deployment, you need to come up with a deployment mechanism by yourself. However, you will probably be able to re-use some of our contents from our metal-roles repository, e.g. the Helm chart for deploying the metal control plane.

TIP

You can use the mini-lab as a template project for your own deployment. It uses the same approach as described in this document.

Metal Control Plane Deployment

The metal control plane is typically deployed in a Kubernetes cluster. Therefore, this document will assume that you have a Kubernetes cluster ready for getting deployed. Even though it is theoretically possible to deploy metal-stack without Kubernetes, we strongly advise you to use the described method because we believe that Kubernetes gives you a lot of benefits regarding the stability and maintainability of the application deployment.

> **TIP**
>
> For metal-stack it does not matter where your control plane Kubernetes cluster is located. You can of course use a cluster managed by a hyperscaler. This has the advantage of not having to setup Kubernetes by yourself and could even become beneficial in terms of fail-safe operation. However, we also describe a solution of how to setup metal-stack with a self-hosted, Autonomous Control Plane cluster. The only requirement from metal-stack is that your partitions can establish network connections to the metal control plane. If you are interested, you can find a reasoning behind this deployment decision here.

Let's start off with a fresh folder for your deployment:

```
mkdir -p metal-stack-deployment
cd metal-stack-deployment
```

At the end of this section we are gonna end up with the following files and folder structures:

```
.
├── ansible.cfg
├── deploy_metal_control_plane.yaml
├── files
│   └── certs
│       ├── ca-config.json
│       ├── ca-csr.json
│       ├── metal-api-grpc
│       │   ├── client.json
│       │   ├── server.json
│       ├── masterdata-api
│       │   ├── client.json
│       │   ├── server.json
│       └── roll_certs.sh
├── inventories
│   ├── control-plane.yaml
│   └── group_vars
│       ├── all
│       │   └── images.yaml
│       └── control-plane
│           ├── common.yaml
│           └── metal.yml
├── generate_role_requirements.yaml
└── roles
    └── ingress-controller
        └── tasks
            └── main.yaml
```

You can already define the `inventories/group_vars/all/images.yaml` file. It contains the metal-stack version you are gonna deploy:

```
---
metal_stack_release_version: v0.22.1
```

Releases and Ansible Role Dependencies

As metal-stack consists of many microservices all having individual versions, we have come up with a releases repository. It contains a YAML file (we often call it release vector) describing the fitting versions of all components for every release of metal-stack.

Ansible role dependencies are also part of a metal-stack release. Therefore, we will now write up a playbook, which dynamically renders a `requirements.yaml` file from the ansible-roles defined in the release repository. The `requirements.yaml` can then be used to resolve the actual role dependencies through Ansible Galaxy. Define the following playbook in `generate_role_requirements.yaml`:

```
---
- name: generate requirements.yaml
  hosts: control-plane
  connection: local
  gather_facts: false
  vars:
    release_vector_url: "https://raw.githubusercontent.com/metal-stack/releases/{{
metal_stack_release_version }}/release.yaml"
  tasks:
    - name: download release vector
      uri:
        url: "{{ release_vector_url }}"
        return_content: yes
      register: release_vector

    - name: write requirements.yaml from release vector
      copy:
        dest: "{{ playbook_dir }}/requirements.yaml"
        content: |
          {% for role_name, role_params in (release_vector.content | from_yaml).get('ansible-
roles').items() %}
          - src: {{ role_params.get('repository') }}
            name: {{ role_name }}
            version: {{ hostvars[inventory_hostname][role_name | lower | replace('-', '_') +
'_version'] | default(role_params.get('version'), true) }}
          {% endfor %}
```

This playbook will always be run before the actual metal-stack deployment and provide you with the proper versions of the Ansible role dependencies.

Inventory

Then, there will be an inventory for the control plane deployment in `inventories/control-plane.yaml` that adds `localhost` to the `control-plane` host group:

```
---
control-plane:
  hosts:
    localhost:
      ansible_python_interpreter: "{{ ansible_playbook_python }}"
```

We do this since we are deploying to Kubernetes and do not need to SSH-connect to any hosts for the deployment (which is what Ansible typically does). This inventory is also necessary to pick up the variables inside `inventories/group_vars/control-plane` during the deployment.

We recommend using the following `ansible.cfg`:

```
[defaults]
retry_files_enabled = false
force_color = true
host_key_checking = false
stdout_callback = yaml
jinja2_native = true
transport = ssh
timeout = 30
force_valid_group_names = ignore

[ssh_connection]
retries=3
ssh_executable = /usr/bin/ssh
```

Most of the properties in there are up to taste, but make sure you enable the Jinja2 native environment as this is needed for some of our roles in certain cases.

Control Plane Playbook

Next, we will define the actual deployment playbook in a file called `deploy_metal_control_plane.yaml`. You can start with the following lines:

```
---
- name: Deploy Control Plane
  hosts: control-plane
  connection: local
  gather_facts: no
  vars:
    setup_yaml:
      - url: https://raw.githubusercontent.com/metal-stack/releases/{{ metal_stack_release_version
}}/release.yaml
        meta_var: metal_stack_release
  roles:
    - name: ansible-common
      tags: always
    - name: ingress-controller
      tags: ingress-controller
    - name: metal-roles/control-plane/roles/prepare
      tags: prepare
    - name: metal-roles/control-plane/roles/nsq
      tags: nsq
    - name: metal-roles/control-plane/roles/metal-db
      tags: metal-db
    - name: metal-roles/control-plane/roles/ipam-db
      tags: ipam-db
    - name: metal-roles/control-plane/roles/masterdata-db
      tags: masterdata-db
```

```
    - name: metal-roles/control-plane/roles/metal
      tags: metal
```

Basically, this playbook does the following:

- Include all the modules, filter plugins, etc. of ansible-common into the play
- Deploys an ingress-controller into your cluster
- Deploys the metal-stack by
  - Running preparation tasks
  - Deploying NSQ
  - Deploying the rethinkdb database for the metal-api (wrapped in a backup-restore-sidecar),
  - Deploying the postgres database for go-ipam (wrapped in a backup-restore-sidecar)
  - Deploying the postgres database for the masterdata-api (wrapped in a backup-restore-sidecar)
  - Applying the metal control plane helm chart

Setup an ingress-controller

As a next step you have to add a task for deploying an ingress-controller into your cluster. nginx-ingress is what we use. If you want to use another ingress-controller, you need to parametrize the metal roles carefully. When you just use ingress-nginx, make sure to also deploy it to the default namespace ingress-nginx.

This is how your `roles/ingress-controller/tasks/main.yaml` could look like:

```
- name: Deploy ingress-controller
  include_role:
    name: ansible-common/roles/helm-chart
  vars:
    helm_repo: "https://helm.nginx.com/stable"
    helm_chart: nginx-ingress
    helm_release_name: nginx-ingress
    helm_target_namespace: ingress-nginx
```

> **TIP**
>
> The ansible-common repository contains very general roles and modules that you can also use when extending your deployment further.

Deployment Parametrization

Now you can parametrize the referenced roles to fit your environment. The role parametrization can be looked up in the role documentation on metal-roles/control-plane. You should not need to define a lot of variables for the beginning as most values are reasonably defaulted. You can start with the following content for `group_vars/control-plane/common.yaml`:

```
---
metal_control_plane_ingress_dns: <your-dns-domain> # if you do not have a DNS entry, you could also
```

```
start with <ingress-ip>.nip.io
```

Providing Certificates

We have several components in our stack that communicate over encrypted gRPC just like Kubernetes components do.

For the very basic setup you will need to create self-signed certificates for the communication between the following components (see architecture document):

- metal-api and masterdata-api (in-cluster traffic communication)
- metal-api and metal-hammer (partition to control plane communication)

Here is a snippet for `files/roll_certs.sh` that you can use for generating your certificates (requires cfssl):

```
#!/usr/bin/env bash
set -eo pipefail

for i in "$@"
do
case $i in
    -t=*|--target=*)
    TARGET="${i#*=}"
    shift
    ;;
    *)
    echo "unknown parameter passed: $1"
    exit 1
    ;;
esac
done

if [ -z "$TARGET" ]; then
    echo "generating ca cert"
    cfssl genkey -initca ca-csr.json | cfssljson -bare ca
    rm *.csr
fi

if [ -z "$TARGET" ] || [ $TARGET == "grpc" ]; then
    pushd metal-api-grpc
    echo "generating grpc certs"
    cfssl gencert -ca=../ca.pem -ca-key=../ca-key.pem -config=../ca-config.json -profile=server
server.json | cfssljson -bare server
    cfssl gencert -ca=../ca.pem -ca-key=../ca-key.pem -config=../ca-config.json -profile=client
client.json | cfssljson -bare client
    rm *.csr
    popd
fi

if [ -z "$TARGET" ] || [ $TARGET == "masterdata-api" ]; then
    pushd masterdata-api
    echo "generating masterdata-api certs"
    rm -f *.pem
    cfssl gencert -ca=../ca.pem -ca-key=../ca-key.pem -config=../ca-config.json -profile=client-
server server.json | cfssljson -bare server
```

```
    cfssl gencert -ca=../ca.pem -ca-key=../ca-key.pem -config=../ca-config.json -profile=client
client.json | cfssljson -bare client
    rm *.csr
    popd
fi
```

Also define the following configurations for `cfssl`:

- `files/certs/ca-config.json`

```
{
  "signing": {
    "default": {
      "expiry": "43800h"
    },
    "profiles": {
      "server": {
        "expiry": "43800h",
        "usages": ["signing", "key encipherment", "server auth"]
      },
      "client": {
        "expiry": "43800h",
        "usages": ["signing", "key encipherment", "client auth"]
      },
      "client-server": {
        "expiry": "43800h",
        "usages": [
          "signing",
          "key encipherment",
          "client auth",
          "server auth"
        ]
      }
    }
  }
}
```

- `files/certs/ca-csr.json`

```
{
  "CN": "metal-control-plane",
  "hosts": [],
  "key": {
    "algo": "rsa",
    "size": 4096
  },
  "names": [
    {
      "C": "DE",
      "L": "Munich",
      "O": "Metal-Stack",
      "OU": "DevOps",
      "ST": "Bavaria"
    }
  ]
}
```

- `files/certs/masterdata-api/client.json`

```
{
  "CN": "masterdata-client",
  "hosts": [""],
  "key": {
    "algo": "ecdsa",
    "size": 256
  },
  "names": [
    {
      "C": "DE",
      "L": "Munich",
      "O": "Metal-Stack",
      "OU": "DevOps",
      "ST": "Bavaria"
    }
  ]
}
```

- `files/certs/masterdata-api/server.json`

```
{
  "CN": "masterdata-api",
  "hosts": [
    "localhost",
    "masterdata-api",
    "masterdata-api.metal-control-plane.svc",
    "masterdata-api.metal-control-plane.svc.cluster.local"
  ],
  "key": {
    "algo": "ecdsa",
    "size": 256
  },
  "names": [
    {
      "C": "DE",
      "L": "Munich",
      "O": "Metal-Stack",
      "OU": "DevOps",
      "ST": "Bavaria"
    }
  ]
}
```

- `files/certs/metal-api-grpc/client.json`

```
{
  "CN": "grpc-client",
  "hosts": [""],
  "key": {
    "algo": "rsa",
    "size": 4096
  },
  "names": [
    {
      "C": "DE",
      "L": "Munich",
      "O": "Metal-Stack",
      "OU": "DevOps",
      "ST": "Bavaria"
    }
```

```
  ]
}
```

- `files/certs/metal-api-grpc/server.json` (**Fill in your control plane ingress DNS here**)

```
{
  "CN": "metal-api",
  "hosts": ["<your-metal-api-dns-ingress-domain>"],
  "key": {
    "algo": "rsa",
    "size": 4096
  },
  "names": [
    {
      "C": "DE",
      "L": "Munich",
      "O": "Metal-Stack",
      "OU": "DevOps",
      "ST": "Bavaria"
    }
  ]
}
```

Running the `roll_certs.sh` bash script without any arguments should generate you the required certificates.

Now Provide the paths to these certificates in `group_vars/control-plane/metal.yaml`:

```
---
metal_masterdata_api_tls_ca: "{{ lookup('file', 'certs/ca.pem') }}"
metal_masterdata_api_tls_cert: "{{ lookup('file', 'certs/masterdata-api/server.pem') }}"
metal_masterdata_api_tls_cert_key: "{{ lookup('file', 'certs/masterdata-api/server-key.pem') }}"
metal_masterdata_api_tls_client_cert: "{{ lookup('file', 'certs/masterdata-api/client.pem') }}"
metal_masterdata_api_tls_client_key: "{{ lookup('file', 'certs/masterdata-api/client-key.pem') }}"

metal_api_grpc_certs_server_key: "{{ lookup('file', 'certs/metal-api-grpc/server-key.pem') }}"
metal_api_grpc_certs_server_cert: "{{  lookup('file', 'certs/metal-api-grpc/server.pem') }}"
metal_api_grpc_certs_client_key: "{{ lookup('file', 'certs/metal-api-grpc/client-key.pem') }}"
metal_api_grpc_certs_client_cert: "{{  lookup('file', 'certs/metal-api-grpc/client.pem') }}"
metal_api_grpc_certs_ca_cert: "{{ lookup('file', 'certs/ca.pem') }}"
```

TIP

For the actual communication between the metal-api and the user clients (REST API, runs over the ingress-controller you deployed before), you can simply deploy a tool like cert-manager into your Kubernetes cluster, which will automatically provide your ingress domains with Let's Encrypt certificates.

Running the Deployment

Finally, it should be possible to run the deployment through a Docker container. Make sure to have the Kubeconfig file of your cluster and set the path in the following command accordingly:

```
export KUBECONFIG=<path-to-your-cluster-kubeconfig>
export METAL_VERSION=v0.22.1
```

Then you can spin up the deployment with docker:

```
docker run --rm -it \
  -v $(pwd):/workdir \
  --workdir /workdir \
  -e KUBECONFIG="${KUBECONFIG}" \
  -e K8S_AUTH_KUBECONFIG="${KUBECONFIG}" \
  -e ANSIBLE_INVENTORY=inventories/control-plane.yaml \
  ghcr.io/metal-stack/metal-deployment-base:${METAL_VERSION} \
  /bin/bash -ce \
    "ansible-playbook obtain_role_requirements.yaml
     ansible-galaxy install -r requirements.yaml
     ansible-playbook deploy_metal_control_plane.yaml"
```

TIP

If you are having issues regarding the deployment take a look at the troubleshoot document. Please give feedback such that we can make the deployment of the metal-stack easier for you and for others!

Providing Images

After the deployment has finished (hopefully without any issues!), you should consider deploying some masterdata entities into your metal-api. For example, you can add your first machine sizes and operating system images. You can do this by further parametrizing the metal role. We will just add an operating system for demonstration purposes. Add the following variable to your `inventories/group_vars/control-plane/common.yaml`:

```
metal_api_images:
- id: firewall-ubuntu-2.0.20201004
  name: Firewall 2 Ubuntu 20201004
  description: Firewall 2 Ubuntu 20201004
  url: http://images.metal-stack.io/metal-os/master/firewall/2.0-ubuntu/20201004/img.tar.lz4
  features:
    - firewall
- id: ubuntu-20.04.20201004
  name: Ubuntu 20.04 20201004
  description: Ubuntu 20.04 20201004
  url: http://images.metal-stack.io/metal-os/master/ubuntu/20.04/20201004/img.tar.lz4
  features:
    - machine
```

Then, re-run the deployment to apply your changes. Our playbooks are idempotent.

INFO

> Image versions should be regularly checked for updates.

Setting up metalctl

You can now verify the existence of the operating system images in the metal-api using our CLI client called metalctl. The configuration for `metalctl` should look like this:

```
# ~/.metalctl/config.yaml
---
current: test
contexts:
  test:
    # the metal-api endpoint depends on your dns name specified before
    # you can look up the url to the metal-api via the kubernetes ingress
    # resource with:
    # $ kubectl get ingress -n metal-control-plane
    url: <metal-api-endpoint>
    # in the future you have to change the HMAC to a strong, random string
    # in order to protect against unauthorized api access
    # the default hmac is "change-me"
    hmac: change-me
```

Issue the following command:

```
$ metalctl image ls
ID                                  NAME                          DESCRIPTION
FEATURES        EXPIRATION      STATUS
ubuntu-19.10.20200331               Ubuntu 19.10 20200331         Ubuntu 19.10 20200331
machine         89d 23h         preview
```

The basic principles of how the metal control plane can be deployed should now be clear. It is now up to you to move the deployment execution into your CI and add things like certificates for the ingress-controller and NSQ.

Setting Up the backup-restore-sidecar

The backup-restore-sidecar can come in very handy when you want to add another layer of security to the metal-stack databases in your Kubernetes cluster. The sidecar takes backups of the metal databases in small time intervals and stores them in a blobstore of a cloud provider. For each database that will be backed up, a lifecycle rule is established. The backup mechanism is deactivated by default and must be activated by the operator. This way your metal-stack setup can even survive the deletion of your Kubernetes control plane cluster (including all volumes getting lost). After re-deploying metal-stack to another Kubernetes clusters, the databases come up with the latest backup data in a matter of seconds.

Encryption can be enabled for the backups by providing an AES-256 encryption key.

Checkout the role documentation of the individual databases to find out how to configure the sidecar properly. You can also try out the mechanism from the backup-restore-sidecar repository.

Auth

metal-stack currently supports two authentication methods:

- user authentication through OpenID Connect (OIDC)
- HMAC auth, typically used for access by technical users (because we do not have service account tokens at the time being)

If you decided to use OIDC, you can parametrize the metal role for this by defining the variable `metal_masterdata_api_tenants` with the following configuration:

```
---
metal_masterdata_api_tenants:
  - meta:
      id: <id>
      kind: Tenant
      apiversion: v1
      version: 0
    name: <name>
    iam_config:
      issuer_config:
        client_id: <client_id>
        url: <oidc_url>
      idm_config:
        idm_type: <type> # "AD" | "UX"
      group_config:
        namespace_max_length: 20
    description: <description>
```

Bootstrapping a Partition

Out-Of-Band-Network

To be able to deploy and maintain a metal-stack partition, you need to bootstrap the Out-Of-Band-Network first. Some considerations must be made to fulfill the requirements of our infrastructure, a partition is designed to be:

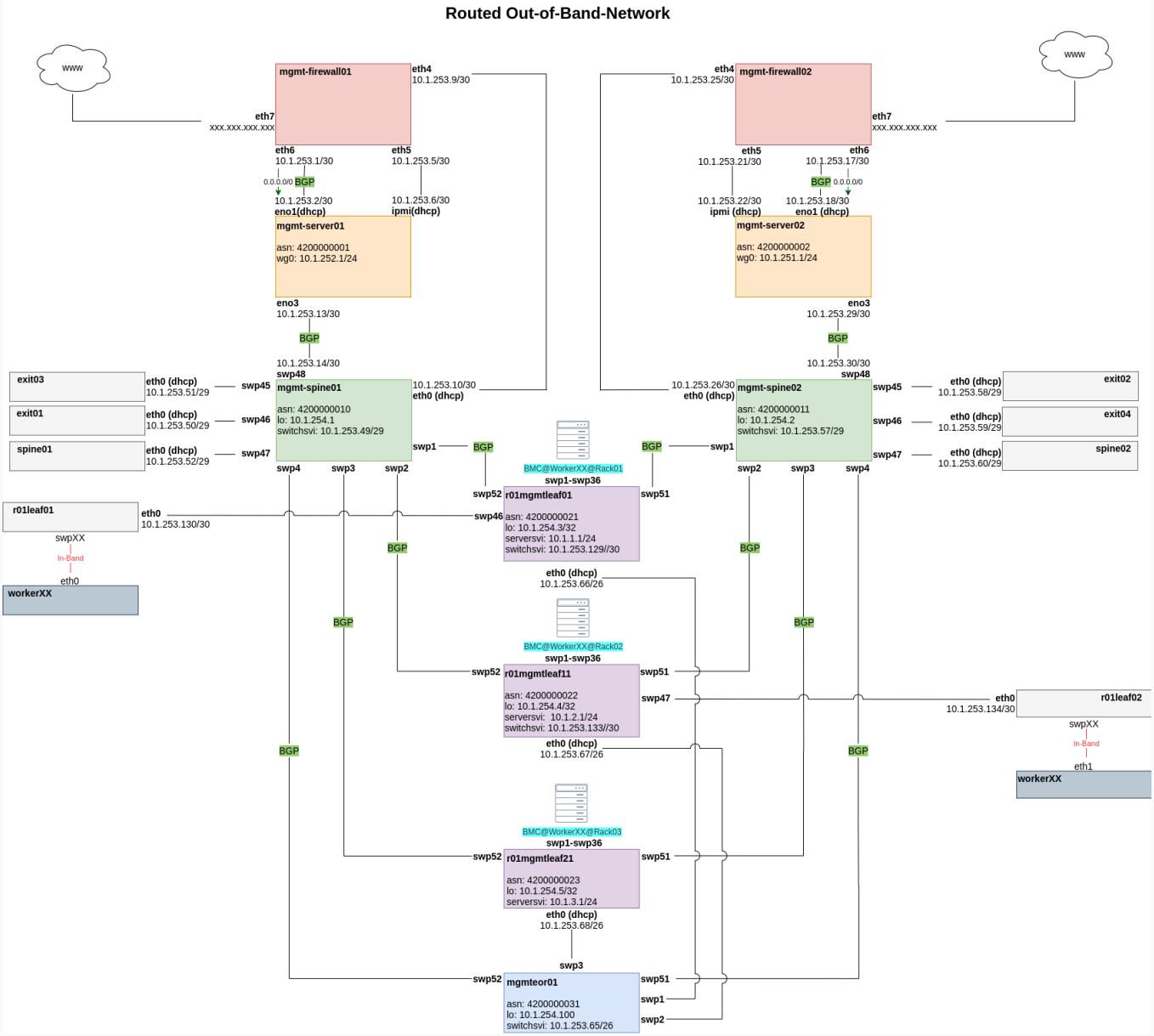
- secure
- fully routable (BGP)
- scalable
- resilient
- deployable via CI/CD jobs
- accessible from the internet from specific IPs

In order to accomplish this task remotely and in a nearly automatic manner, you have to bootstrap the components in this order:

1. management firewalls
2. management servers
3. management spines

4. management leaves
5. leaves, spines and exits

This document assumes that all cabling is done. Here is a quick overview of the architecture:



Management Firewalls

As you can see, the management firewalls are the first bastion hosts in a partition to provide access to our infrastructure. There are two of them in each partition to guarantee high availability and load balancing. The very first configuration of these routers has to be done manually to solve the chicken and egg problem that you need the management firewalls in place to deploy the partition. Manually means that we generate a configuration template with ansible that we deploy with copy/paste, and the load, through the machine console. Once the management server has been deployed, we are able to deploy this configuration via CI runner and ansible. For this you need the user and the ssh-key, which is deployed with the configuration file mentioned above. The Edgerouters has to fulfill some requirements including:

- provide and restrict access to the Out-Of-Band-Network from the internet with a firewall ruleset
- provide destination NAT to the management server and its IPMI interface
- provide Onie Boot and ztp via DHCP options for the management spine
- provide DHCP management addresses for management spine, management server and ipmi interface of the management server
- Hairpin-NAT for the management server to access itself via its public IP, needed by the CI runner to delegate CI Jobs.
- propagate a default gateway via BGP

Management Servers

The second bastion hosts are the management servers. They are the main bootstrapping components of the Out-Of-Band-Network. They also act as jump hosts for all components in a partition. Once they are installed and deployed, we are able to bootstrap all the other components. To bootstrap the management servers, we generate an ISO image which will automatically install an OS and an ansible user with ssh keys. It is preconfigured with a preseed file to allow an unattended OS installation for our needs. This is why we need remote access to the IPMI interface of the management servers: The generated ISO is attached via the virtual media function of the BMC. After that, all we have to do is boot from that virtual CD-ROM and wait for the installation to finish. Deployment jobs (Gitlab-CI) in a partition are delegated to the appropriate management servers, therefore we need a CI runner active on each management server.

After the CI runner has been installed, you can trigger your Playbooks from the the CI. The Ansible-Playbooks have to make sure that these functionalities are present on the management servers:

- Prometheus and exporters
- CI runner
- metal-bmc
- image-cache
- simple webserver to provide images
- Onie Boot and ZTP
- DHCP addresses for ipmi interfaces of the workers
- DHCP addresses for switches

Management Spines

> **TIP**
>
> If you are using SONiC switches, you should make use of Zero Touch Provisioning and Onie Boot

The purpose of these switches is to connect the management interfaces of all switches to the management servers. The management spine's own management interface is connected to the management firewall for the bootstrapping of the management spine itself. The management firewall will provide a DHCP address and DHCP options to start SONiC's Zero Touch Provisioning; the images for all switches are downloaded from the management server (nginx container). Each management leaf is connected to both management spines to provide redundant connectivity to both management servers. BGP is used as a routing protocol such that, when a link goes down, an alternate path is used. In the picture above you can see that there are

also switch management interfaces connected to the management spine. This has to be done so that we can bootstrap these switches; the management spine relays the DHCP requests from these switches to the management servers so that they are able to Onie Boot and get their ZTP scripts.

Management Leaves

All workers have to be connected with their IPMI/BMC interface to the management leaves to get DHCP addresses from the management server. The management leaves are relaying those DHCP requests to the management server which will answer the requests and provide IPs from a given range. The management interfaces of the management leaves also have to be reachable from the management server, and need to get their IP address via DHCP for the bootstrapping process.

In the example setup, these interfaces are connected to an end-of-row-switch which aggregates them and connects them to the management spines with a fiber-optics connection. If you can reach the management spines from the management leaves with copper cables, you do not need the end of row switch. After the initial bootstrapping, the management interfaces of the management leaves continue to be used for access to the switches' command line, and for subsequent OS updates. (update=reset+bootrap+deployment)

Partition Deployment

Gardener with metal-stack

If you want to deploy metal-stack as a cloud provider for Gardener, you should follow the regular Gardener installation instructions and setup a Gardener cluster first. It's perfectly fine to setup the Gardener cluster in the same cluster that you use for hosting metal-stack.

You can find installation instructions for Gardener on the Gardener website beneath docs. metal-stack is an out-of-tree provider and therefore you will not find example files for metal-stack resources in the Gardener repositories. The following list describes the resources and components that you need to deploy into the Gardener cluster in order to make Gardener work with metal-stack:

> **WARNING**
>
> The following list assumes you have Gardener installed in a Kubernetes cluster and that you have a basic understanding of how Gardener works. If you need further help with the following steps, you can also come and ask in our Slack channel.

1. Deploy the validator from the gardener-extension-provider-metal repository to your cluster via Helm
2. Add a cloud profile called `metal` containing all your machine images, machine types and regions (region names can be chosen freely, the zone names need to match your partition names) together with our metal-stack-specific provider config as defined here
3. Register the gardener-extension-provider-metal controller by deploying the controller-registration into your Gardener cluster, parametrize the embedded chart in the controller registration's values section if necessary (this is the corresponding values file)

4. metal-stack does not provide an own backup storage infrastructure for now. If you want to enable ETCD backups (which you should do because metal-stack also does not have persistent storage out of the box, which makes these backups even more valuable), you should deploy an extension-provider of another cloud provider and configure it to only reconcile the backup buckets (you can reference this backup infrastructure used for the metal shoot in the shoot spec)
5. Register the os-extension-provider-metal controller by deploying the controller-registration into your Gardener cluster, this controller can transform the operating system configuration from Gardener into Ignition user data
6. You need to use the Gardener's networking-calico controller for setting up shoot CNI, you will have to put specific provider configuration into the shoot spec to make it work with metal-stack:

```
networking:
  type: calico
  # we can peer with the frr within 10.244.0.0/16, which we do with the metallb
  # the networks for the shoot need to be disjunct with the networks of the seed, otherwise the
VPN connection will not work properly
  # the seeds are typically deployed with podCIDR 10.244.128.0/18 and serviceCIDR
10.244.192.0/18
  # the shoots are typically deployed with podCIDR 10.244.0.0/18 and serviceCIDR 10.244.64.0/18
  pods: 10.244.0.0/18
  services: 10.244.64.0/18
  providerConfig:
    apiVersion: calico.networking.extensions.gardener.cloud/v1alpha1
    kind: NetworkConfig
    backend: vxlan
    ipv4:
      pool: vxlan
      mode: Always
      autoDetectionMethod: interface=lo
    typha:
      enabled: false
```

7. For your seed cluster you will need to provide the provider secret for metal-stack containing the key `metalAPIHMac`, which is the API HMAC to grant editor access to the metal-api
8. Checkout our current provider configuration for infrastructure and control-plane before deploying your shoot

> **TIP**
>
> We are officially supported by Gardener dashboard. The dashboard can also help you setting up some of the resources mentioned above.

Maintenance

Update Policy

For new features and breaking changes we create a new minor release of metal-stack. For every minor release we present excerpts of the changes in a corresponding blog article published on metal-stack.io.

It is not strictly necessary to cycle through the patch releases if you depend on the pure metal-stack components. However, it is important to go through all the patch releases and apply all required actions from the release notes. Therefore, we recommend to just install every patch release one by one in order to minimize possible problems during the update process.

In case you depend on the Gardener integration, especially when using metal-stack roles for deploying Gardener, we strongly recommend installing every patch release version. We increment our Gardener dependency version by version following the Gardener update policy. Jumping versions may lead to severe problems with the installation and should only be done if you really know what you are doing.

> **INFO**
>
> If you use the Gardener integration of metal-stack do not skip any patch releases. You may skip patch releases if you depend on metal-stack only, but we recommend to just deploy every patch release one by one for the best possible upgrade experience.

Releases

Before upgrading your metal-stack installation, review the release notes carefully - they contain important information on required pre-upgrade actions and notable changes. These notes are currently shared via a dedicated Slack channel and are also available in the release on GitHub. Once you are prepared, you can deploy a new metal-stack version by updating the `metal_stack_release_version` variable in your Ansible configuration and trigger the corresponding deployment jobs in your CI. metal-stack offers prebuilt system images for firewalls and worker machines, which can be downloaded from `images.metal-stack.io`. In offline or air-gapped setups, these images must either be synced into the partition-local image-cache after they were added to the metal-api or be manually downloaded in advance and uploaded to your local S3-compatible storage. Ensure that the image paths and metadata are correctly maintained so the system can retrieve them during provisioning. If you are using metal-stack in combination with Gardener and you do not run pre-production stages, we advise running some basic functional tests after upgrading metal-stack to ensure the installation is in a fully functional state (e.g. reconciling a bunch of shoot clusters with evaluation purpose, creating and deleting a shoot cluster). metal-images for firewalls and worker nodes follow independent release cycles, typically driven by the need for security patches or system updates. When new images are made available, the machines must be re-provisioned to apply the updates. When using metal-stack in a Kubernetes context, this results in a rolling update of the cluster worker groups. In a Gardener setup, image updates can be triggered by referencing the new image in the shoot spec. Because all

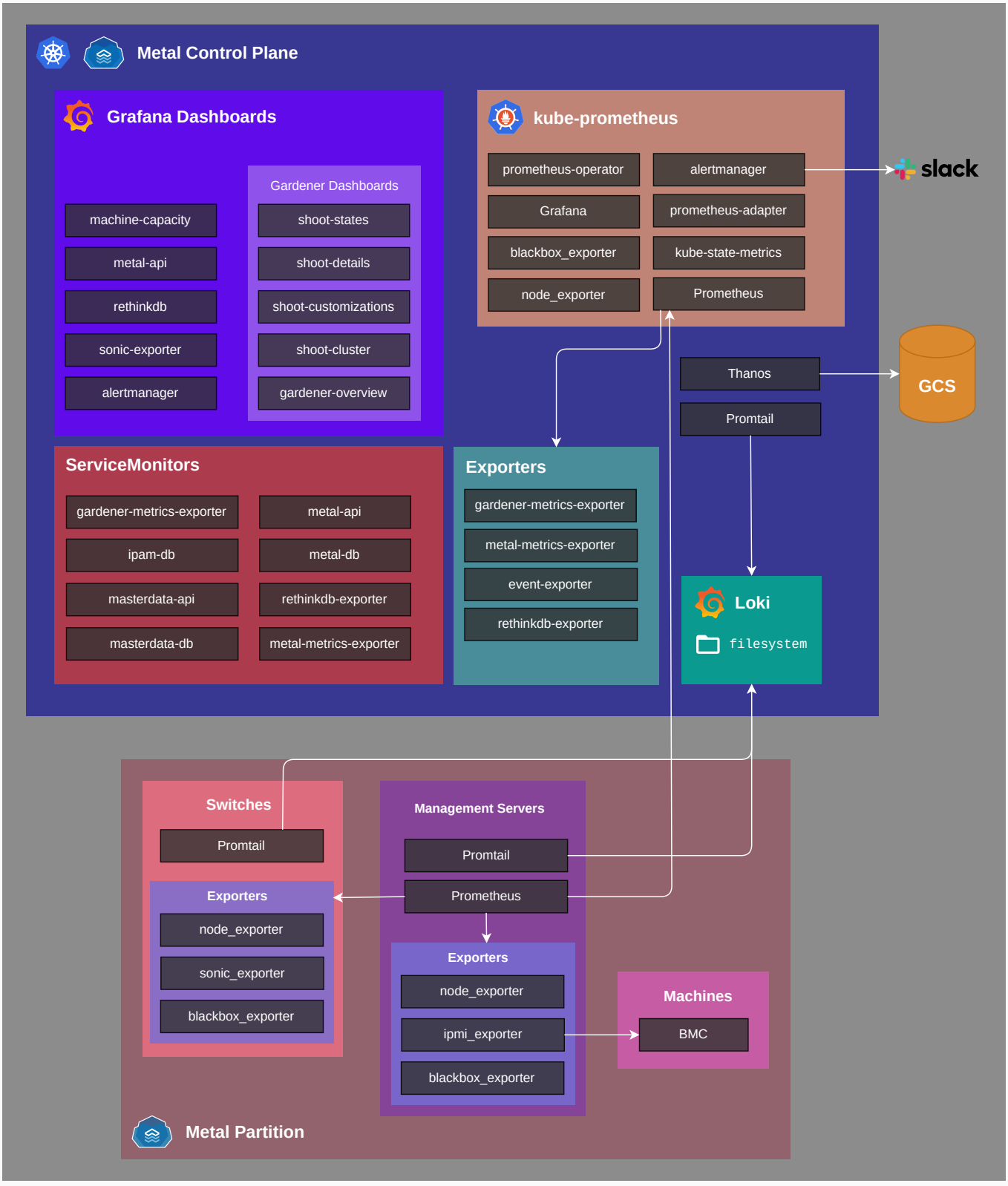
outbound traffic passes through the firewall node, this results in a short downtime of around 30 seconds. This interruption only occurs if the firewall image has actually changed. The process works as follows: a new firewall node is provisioned and configured in parallel with the existing one. Once setup is complete, traffic is switched over to the new node, and the old firewall is then decommissioned. This minimizes disruption while ensuring a seamless transition. The worker nodes are rolled out one after the other and, if possible, the containers are redistributed to the machines that are still available. However, for unclustered stateful workloads like databases, temporary disruptions may occur during node restarts.

Rollback

metal-stack employs forward-only database migrations (e.g., for RethinkDB), and each release undergoes thorough integration testing. However, rollback procedures are not included in test coverage. To maintain data integrity and system reliability, rolling back a full release is not supported and strongly discouraged. In the event of issues after an upgrade, it is possible to downgrade specific components rather than reverting the entire system.

Monitoring the metal-stack

Overview



Logging

Logs are being collected by Promtail and pushed to a Loki instance running in the control plane. Loki is deployed in monolithic mode and with storage type `'filesystem'`. You can find all logging related configuration parameters for the control plane in the control plane's logging role.

In the partitions, Promtail is deployed inside a systemd-managed Docker container. Configuration parameters can be found in the partition's promtail role. Which hosts Promtail collects from can be configured via the `prometheus_promtail_targets` variable.

Monitoring

For monitoring we deploy the kube-prometheus-stack and a Thanos instance in the control plane. Metrics for the control plane are supplied by

- `metal-metrics-exporter`
- `rethindb-exporter`
- `event-exporter`
- `gardener-metrics-exporter`

To query and visualize logs, metrics and alerts we deploy several grafana dashboards to the control plane:

- `grafana-dashboard-alertmanager`
- `grafana-dashboard-machine-capacity`
- `grafana-dashboard-metal-api`
- `grafana-dashboard-rethinkdb`
- `grafana-dashboard-sonic-exporter`

and also some gardener related dashboards:

- `grafana-dashboard-gardener-overview`
- `grafana-dashboard-shoot-cluster`
- `grafana-dashboard-shoot-customizations`
- `grafana-dashboard-shoot-details`
- `grafana-dashboard-shoot-states`

The following `ServiceMonitors` are also deployed:

- `gardener-metrics-exporter`
- `ipam-db`
- `masterdata-api`
- `masterdata-db`
- `metal-api`
- `metal-db`
- `rethinkdb-exporter`
- `metal-metrics-exporter`

All monitoring related configuration parameters for the control plane can be found in the control plane's monitoring role.

Partition metrics are supplied by

- `node-exporter`
- `blackbox-exporter`
- `ipmi-exporter`
- `sonic-exporter`
- `metal-core`
- `frr-exporter`

and scraped by Prometheus. For each of these exporters, the target hosts can be defined by

- `prometheus_node_exporter_targets`
- `prometheus_blackbox_exporter_targets`
- `prometheus_frr_exporter_targets`
- `prometheus_sonic_exporter_targets`
- `prometheus_metal_core_targets`
- `prometheus_frr_exporter_targets`

Alerting

In addition to Grafana, alerts can optionally be sent to a Slack channel. For this to work, at least a valid `monitoring_slack_api_url` and a `monitoring_slack_notification_channel` must be specified. For further configuration parameters refer to the monitoring role. Alerting rules are defined in the rules directory of the partition's prometheus role.

Troubleshooting

This document summarizes help when something goes wrong and provides advice on debugging the metal-stack in certain situations.

Of course, it is also advisable to check out the issues on the Github projects for help.

If you still can't find a solution to your problem, please reach out to us and our community. We have a public Slack Channel to discuss problems, but you can also reach us via mail. Check out metal-stack.io for contact information.

Deployment

Ansible fails when the metal control plane helm chart gets applied

There can be many reasons for this. Since you are deploying the metal control plane into a Kubernetes cluster, the first step should be to install kubectl and check the pods in your cluster. Depending on the metal-stack version and Kubernetes cluster, your control-plane should look something like this after the deployment (this is in a Kind cluster):

```
kubectl get pod -A
NAMESPACE            NAME                                         READY   STATUS      RESTARTS
AGE
ingress-nginx        nginx-ingress-controller-56966f7dc7-khfp9    1/1     Running     0
2m34s
kube-system          coredns-66bff467f8-grn7q                     1/1     Running     0
2m34s
kube-system          coredns-66bff467f8-n7n77                     1/1     Running     0
2m34s
kube-system          etcd-kind-control-plane                      1/1     Running     0
2m42s
kube-system          kindnet-4dv7m                                1/1     Running     0
2m34s
kube-system          kube-apiserver-kind-control-plane            1/1     Running     0
2m42s
kube-system          kube-controller-manager-kind-control-plane   1/1     Running     0
2m42s
kube-system          kube-proxy-jz7kp                             1/1     Running     0
2m34s
kube-system          kube-scheduler-kind-control-plane            1/1     Running     0
2m42s
local-path-storage   local-path-provisioner-bd4bb6b75-cwfb7       1/1     Running     0
2m34s
metal-control-plane  ipam-db-0                                    2/2     Running     0
2m31s
metal-control-plane  masterdata-api-6dd4b54db5-rwk45             1/1     Running     0
33s
metal-control-plane  masterdata-db-0                            2/2     Running     0
2m29s
metal-control-plane  metal-api-998cb46c4-jj2tt                   1/1     Running     0
33s
```

```
metal-control-plane   metal-api-initdb-r9sc6                    0/1     Completed   0
2m24s
metal-control-plane   metal-api-liveliness-1590479940-brhc7     0/1     Completed   0          6s
metal-control-plane   metal-console-7955cbb7d7-p6hxp            1/1     Running     0
33s
metal-control-plane   metal-db-0                                2/2     Running     0
2m34s
metal-control-plane   nsq-lookupd-5b4ccbfb64-n6prg              1/1     Running     0
2m34s
metal-control-plane   nsqd-6cd87f69c4-vtn9k                     2/2     Running     0
2m33s
```

If there are any failing pods, investigate those and look into container logs. This information should point you to the place where the deployment goes wrong.

> **INFO**
>
> Sometimes, you see a helm errors like "no deployed releases" or something like this. When a helm chart fails after the first deployment it could be that you have a chart installation still pending. Also, the control plane helm chart uses pre- and post-hooks, which creates jobs that helm expects to be completed before attempting another deployment. Delete the helm chart (use Helm 3) with `helm delete -n metal-control-plane metal-control-plane` and delete the jobs in the `metal-control-plane` namespace before retrying the deployment.

In the mini-lab the control-plane deployment fails because my system can't resolve api.172.17.0.1.nip.io

The control-plane deployment returns an error like this:

```
deploy-control-plane | fatal: [localhost]: FAILED! => changed=false
deploy-control-plane |   attempts: 60
deploy-control-plane |   content: ''
deploy-control-plane |   elapsed: 0
deploy-control-plane |   msg: 'Status code was -1 and not [200]: Request failed: <urlopen error
[Errno -5] No address associated with hostname>'
deploy-control-plane |   redirected: false
deploy-control-plane |   status: -1
deploy-control-plane |   url: http://api.172.17.0.1.nip.io:8080/metal/v1/health
deploy-control-plane |
deploy-control-plane | PLAY RECAP
*********************************************************************
deploy-control-plane | localhost                  : ok=29   changed=4    unreachable=0    failed=1
skipped=7    rescued=0    ignored=0
deploy-control-plane |
deploy-control-plane exited with code 2
```

Some home routers have a security feature that prevents DNS Servers to resolve anything in the router's local IP range (DNS-Rebind-Protection).

You need to add an exception for `nip.io` in your router configuration or add `127.0.0.1 api.172.17.0.1.nip.io` to your `/etc/hosts`.

FritzBox

`Home Network -> Network -> Network Settings -> Additional Settings -> DNS Rebind Protection -> Host name exceptions -> nip.io`

Operations

Fixing Machine Issues

The `metalctl machine issues` command gives you an overview over machines in your metal-stack environment that are in an unusual state.

> **TIP**
>
> Machines that are known not to function properly, should be locked through `metalctl machine lock` and annotated with a description of the problem. This way, you can mark machine for replacement without being in danger of having a user allocating the faulty machine.

In the following sections, you can look up the machine issues that are returned by `metalctl` and find out how to deal with them properly.

no-event-container

Every machine in the metal-stack database usually has a corresponding event container where provisioning events are stored. This database entity gets created lazily as soon as a machine is registered by the metal-hammer or a provisioning event for the machine arrives at the metal-api.

When there is no event container, this means that the machine has never registered nor received a provisioning event. As an operator you should evaluate why this machine is not booting into the metal-hammer.

This issue is special in a way that it prevents other issues from being evaluated for this machine because the issue calculation usually requires information from the machine event container.

no-partition

When a machine has no partition, the metal-hammer has not yet registered the machine at the metal-api. Instead, the machine was created through metal-stack's event machinery, which does not have a lot of information about a machine (e.g. a PXE boot event was reported from the pixiecore), or just by the metal-bmc which discovered the machine through DHCP.

This can usually happen on the very first boot of a machine and the machine's hardware is not supported by metal-stack, leading to the metal-bmc being unable to report BMC details to the metal-api (a metal-bmc report sets the partition id of a machine) and the metal-hammer not finishing the machine registration phase.

To resolve this issue, you need to identify the machine in your metal-stack partition that emits PXE boot events and find the reason why it is not properly booting into the metal-hammer. The console logs of this machine should enable you to find out the root cause.

liveliness-dead

For machines without an allocation, the metal-hammer consistently reports whether a machine is still being responsive or not. When the liveliness is `Dead`, there were no events received from this machine for longer than ~5 minutes.

Reasons for this can be:

- The network connection between the partition and metal-stack control plane is interrupted
- The machine was removed from your data center
- The machine has changed its UUID metal-hammer#52
- The machine is turned off
- The machine hangs / freezes
- The machine booted to BIOS or UEFI shell and does not try to PXE boot again
- The issue only appears temporarily
  - The machine takes longer than 5 minutes for the reboot
  - The machine is performing a firmware upgrade, which usually takes longer than 5 minutes to succeed

> **INFO**
>
> In order to minimize maintenance overhead, a machine which is dead for longer than an hour will be rebooted through the metal-api.
>
> In case you want to prevent this action from happening for a machine, you can lock the machine through `metalctl machine lock`.

If the machine is dead for a long time and you are sure that it will never come back, you can clean up the machine through `metalctl machine rm --remove-from-database`.

liveliness-unknown

For machines that are allocated by a user, the ownership has gone over to this user and as an operator you cannot access the machine anymore. This makes it harder to detect whether a machine is in a healthy state or not. Typically, all official metal-stack OS images deploy an LLDP daemon, that consistently emits alive messages. These messages are caught by the metal-core and turned into a `Phoned Home` event. Internally, the metal-api uses these events as an indicator to decide whether the machine is still responsive or not.

When the LLDP daemon stopped sending packages, the reasons are identical to those of dead machines. However, it's not possible anymore to decide whether the user is responsible for reaching this state or not.

In most of the cases, there is not much that can be done from the operator's perspective. You will need to wait for the user to report an issue with the machine. When you do support, you can use this issue type to quickly identify this machine.

liveliness-not-available

This is more of a theoretical issue. When the machine liveliness is not available check that the Kubernetes `CronJob` in the metal-stack control plane for evaluating the machine liveliness is running regularly and not containing error logs. Make the machine boot into the metal-hammer and this issue should not appear.

failed-machine-reclaim

If a machine remains in the `Phoned Home` state without having an allocation, this indicates that the metal-bmc was not able to put the machine back into PXE boot mode after `metalctl machine rm`. The machine is still running the operating system and it does not return back into the allocatable machine pool. Effectively, you lost a machine in your environment and no-one pays for it. Therefore, you should resolve this issue as soon as possible.

In bad scenarios, when the machine was a firewall, the machine can still reach the internet through the PXE boot network and also attract traffic, which it cannot route anymore inside the tenant VRF. This can cause traffic loss inside a tenant network.

In most of the cases, it should be sufficient to run another `metalctl machine rm` on this machine in order to retry booting into PXE mode. If this still does not succeed, you can boot the machine into the BIOS and manually and change the boot order to PXE boot. This should force booting the metal-hammer again and add the machine back into your pool of allocatable machines.

For further reference, see metal-api#145.

crashloop

Under bad circumstances, a machine diverges from its typical machine lifecycle. When this happens, the internal state-machine of the metal-api detects that the machine reboots unexpectedly during the provisioning phase. It is likely that the machine has entered a crash loop where it PXE boots again and again without the machine ever becoming usable.

Reasons for this can be:

- The machine's hardware is not supported and the metal-hammer crashes during the machine discovery
- The machine registration fails through the metal-hammer because an orphaned / dead machine is still present in the metal-api's data base. The machine is connected to the same switch ports that were used by the orphaned machine. In this case, you should clean up the orphaned machine through `metalctl machine rm --remove-from-database`.

Please also consider console logs of the machine for investigating the issue.

The incomplete cycle count is reset as soon as the machine reaches `Phoned Home` state or there is a `Planned Reboot` of the machine (planned reboot is also done by the metal-hammer once a day in order to reboot with the latest version).

last-event-error

The machine had an error during the provisioning lifecycle recently or events are arriving out of order at the metal-api. This can be an interesting hint for the operator that something during machine provisioning went wrong. You can look at the error through `metalctl machine describe` or `metalctl machine logs`.

This error will disappear after a certain time period from `machine issues`. You can still look up the error as described above.

asn-not-unique

This issue was introduced by a bug in earlier versions of metal-stack and was fixed in PR105

To resolve the issue, you need to recreate the firewalls that use the same ASN.

bmc-without-mac

The metal-bmc is responsible to report connection data for the machine's BMC.

If it's uncapable of discovering this information, your hardware might not be supported. Please investigate the logs of the metal-bmc to find out what's going wrong with this machine.

bmc-without-ip

The metal-bmc is responsible to report connection data for the machine's BMC.

If it's uncapable of discovering this information, your hardware might not be supported. Please investigate the logs of the metal-bmc to find out what's going wrong with this machine.

bmc-no-distinct-ip

The metal-bmc is responsible to report connection data for the machine's BMC.

When there is no distinct IP address for the BMC, it can be that an orphaned machine used this IP in the past. In this case, you need to clean up the orphaned machine through `metalctl machine rm --remove-from-database`.

bmc-info-outdated

The metal-bmc is responsible to report bmc details for the machine's BMC.

When the metal-bmc was not able to fetch the bmc info for longer than 20 minutes, something is wrong with the BMC configuration of the machine. This can be caused by one of the following reasons:

- Wrong password for the root user is configured in the BMC
- ip address of the BMC is either wrong or not present
- the device on the given ip address is not a machine, maybe a switch or a management component which is not managed by the metal-api

In either case, please check the logs for the given machine UUID on the metal-bmc for further details. Also check that the metal-bmc is configured to only consider BMC IPs in the range they are configured from the DHCP server in the partition. This prevents grabbing unrelated BMCs.

A machine has registered with a different UUID after reboot

metal-stack heavily relies on steady machine UUIDs as the UUID is the primary key of the machine entity in the metal-api.

For further reference also see metal-stack/metal-hammer#52.

Reasons

There are some scenarios (can be vendor-specific), which can cause a machine UUID to change over time, e.g.:

- When the UUID partly contains of a network card's mac address, it can happen when:
  - Exchanging network cards
  - Disabling network cards through BIOS
- Changing the UUID through vendor-specific CLI tool

Solution

1. After five minutes, the orphaned machine UUID will be marked dead (💀) because machine events will be sent only to the most recent UUID
2. Identify the dead machine through `metalctl machine ls`
3. Remove the dead machine forcefully with `metalctl machine rm --remove-from-database --yes-i-really-mean-it <uuid>`

Fixing Switch Issues

switch-sync-failing

For your network infrastructure it is key to adapt to new configuration. In case this sync process fails for more than 10 minutes, it is likely to require manual investigation.

Depending on your switch operating system, the error sources might differ a lot. Try to connect to your switch using the console or ssh and investigate the logs. Check if the hard drive is full.

Switch Replacement and Migration

There are two mechanisms to replace an existing switch with a new one, both of which will transfer existing VRF configuration and machine connections from one switch to another. Due to the redundance of the CLOS topology, a switch replacement can be performed without downtime.

Replacing a Switch

If the new switch should have the same ID as the old one you should perform a switch replacement. To find detailed information about the procedure of a switch replacement use `metalctl switch replace --help`. Basically, what you need to do is mark the switch for replacement via `metalctl switch replace`, then physically replace the switch with the new one and configure it. The last step is to deploy metal-core on the switch. Once metal-core registers the new switch at the metal-api, the old switches configuration and machine connections will be transferred to the new one. Note that the replacement only works if the new switch has the same ID as the old one. Otherwise metal-core will simply register a new switch and leave the old one untouched.

Migrating from one Switch to another

If the new switch should not or cannot have the same ID as the old one, then the `switch migrate` command can be used to achieve the same result as a switch replacement. Perform the following steps:

1. Leave the old switch in place.
2. Install the new switch in the rack without connecting it to any machines yet.
3. Adjust the metal-stack deployment in the same way as for a switch replacement.
4. Deploy metal-core on the new switch and wait for it to register at the metal-api. Once the switch is registered it will be listed when you run `metalctl switch ls`.
5. Run `metalctl switch migrate <old-switch-id> <new-switch-id>`.
6. Disconnect all machines from the old switch and connect them to the new one.

In between steps 5 and 6 there is a mismatch between the switch-machine-connections known to the metal-api and the real connections. Since the metal-api learns about the connections from what a machine reports during registration, a machine registration that occurs in between steps 5 and 6 will result in a condition that looks somewhat broken. The metal-api will think that a machine is connected to three switches. This, however, should not cause any problems. Just move on to step 6 and delete the old switch from the metal-api afterwards. If the case just described really occurs, then `metalctl switch delete <old-switch-id>` will throw an error, because deleting a switch with existing machine connections might be dangerous. If, apart from that, the migration was successful, then the old switch can be safely deleted with `metalctl switch delete <old-switch-id> --force`.

Preconditions for Migration and Replacement

An invariant that must be satisfied throughout is that the switch ports a machine is connected to must match, i.e. a machine connected to `Ethernet0` on switch 1 must be connected to `Ethernet0` on switch 2 etc. Furthermore, the breakout configurations of both switches must match and the new switch must contain at least all of the old switch's interfaces.

Migrating from Cumulus to Edgecore SONiC

Both migration and replacement can be used to move from Cumulus to Edgecore SONiC (or vice versa). Migrating to or from Broadcom SONiC or mixing Broadcom SONiC with Cumulus or Edgecore SONiC is not supported.

Connect a Machine to Another Switch Pair

As soon as a machine was connected to the management network and a pair of leaf switches, and the metal-hammer successfully registered the machine at the metal-api after PXE boot, the `switch` entity in metal-stack contains the machine ID in a data structure called *machine connections*.

In case you would like to wire this machine to another pair of switches inside this partition, the metal-api would prevent the machine registration because it finds that the machine is already connected to other switches in this partition.

To resolve this state, the approach for recabling a machine works as follows:

1. Free the machine if it still has an allocation.
2. Reconnect the machine to the new switch pair.
3. Leave the machine turned off or turn it off and wait until the machine reaches the dead state (💀) in the metal-api.
4. Delete the machine through `metalctl machine delete <id> --remove-from-database --yes-i-really-mean-it`. This cleans up the existing machine connections, too.
5. The machine will soon show up again because the metal-bmc discovers it through the DHCP address obtained by the machine BMC.
6. Power on the machine again and let the metal-hammer register the machine.

Architecture

The metal-stack is a compound of microservices predominantly written in Golang.

This page gives you an overview over which microservices exist, how they communicate with each other and where they are deployed.

Target Deployment Platforms

For our environments, we chose to deploy the metal-stack into a Kubernetes cluster. This means that also our entire installation was developed for metal-stack being run on Kubernetes. Running applications on Kubernetes gives you a lot of benefits regarding ease-of-deployment, scalability, reliability and so on.

However, very early we decided that we do not want to depend on technical Kubernetes functionality with our software (i.e. we did not implement the stack "kube-native" by using controllers and Kubernetes CRDs and things like that). With the following paragraph we want to point out the reasoning behind this "philosophical" decision that may sound conservative at first glance. But not relying on Kubernetes technology:

- Makes deployments of the stack without Kubernetes theoretically possible.
  - We believe that cloud providers should be able to act beneath Kubernetes
  - This way it is possible to use metal-stack for providing your own Kubernetes offering without relying on Kubernetes yourself (breaks the chicken-egg problem)
- Follows an important claim in microservice development: "Be agnostic to your choice of technology"
  - For applications that are purely made for being run on Kubernetes, it does not matter to rely on this technology (we even do the same a lot with our applications that integrate the metal-stack with Gardener) but as soon as you start using things like the underlying reconciliation abilities (which admittedly are fanstatic) you are locking your code into a certain technology
  - We don't know what comes after Kubernetes but we believe that a cloud offering should have the potential to survive a choice of technology
  - By this decision we ensured that we can migrate the stack to another future technology and survive the change

One more word towards determining the location for your metal control plane: It is not strictly required to run the control plane inside the same data center as your servers. It even makes sense not to do so because this way you can place your control plane and your servers into a different failure domains, which makes your installation more robust to data center meltdown. Externally hosting the control plane brings you up and running quickly plus having the advantage of higher security through geo-distribution.

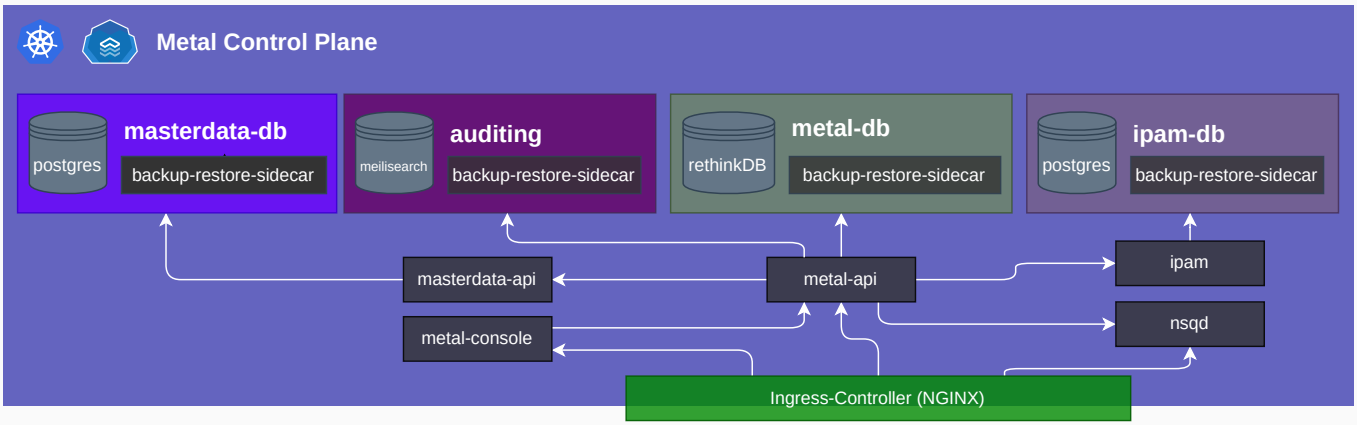
Metal Control Plane

The foundation of the metal-stack is what we call the *metal control plane*.

The control plane contains a couple of essential microservices for the metal-stack including:

- **metal-api** The API to manage control plane resources like machines, switches, operating system images, machine sizes, networks, IP addresses and more. The exposed API is an old-fashioned REST API with different authentication methods. The metal-api stores the state of these entities in a RethinkDB database. The metal-api also has its own IP address management (go-ipam), which writes IP address and network allocations into a PostgreSQL backend.
- **masterdata-api** Manages tenant and project entities, which can be described as entities used for company-specific resource separation and grouping. Having these "higher level entities" managed by a separate microservice was a design choice that allows to re-use the information by other microservices without having them to know the metal-api at all. The masterdata gets persisted in a dedicated PostgreSQL database.
- **metal-console** Provides access for users to a machine's serial console via SSH. It can be seen as an optional component.
- **nsq** A message queuing system (not developed by the metal-stack) used for decoupling microservices and distributing tasks.

The following figure shows the relationships between these microservices:



> Figure 1: The metal control plane deployed in a Kubernetes environment with an ingress-controller exposing additional services via service exposal.

Some notes on this picture:

- Users can access the metal-api with the CLI client called metalctl.
- You can programmatically access the metal-api with client libraries (e.g. metal-go).
- Our databases are wrapped in a specially built backup-restore-sidecar, which is consistently backing up the databases in external blob storage.
- The metal-api can be scaled out using replicas when being deployed in Kubernetes.

Partitions

A *partition* is our term for describing hardware in the data center controlled by the metal-stack with all the hardware participating in the same network topology. Being in the same network topology causes the hardware inside a partition to build a failure domain. Even though the network topology for running the metal-stack is required to be redundant by design, you should consider setting up multiple partitions. With multiple partitions it is possible for users to maintain availability of their applications by spreading them across the partitions. Installing partitions in multiple data centers would be even better in regards of fail-safe application performance, which would even tolerate the meltdown of a data center.

> **TIP**
>
> In our setups, we encode the name of a region and a zone name into our partition names. However, we do not have dedicated entities for regions and zones in our APIs.
>
> A **region** is a geographic area in which data centers are located.
>
> **Zones** are geographic locations in a region usually in different fire compartments. Regions can consist of several zones.
>
> A zone can consist of several **partitions**. Usually, a partition spans a rack or a group of racks.

We strongly advise to group your hardware into racks that are specifically assembled for running metal-stack. When using modular rack design, the amount of compute resources of a partition can easily be extended by adding more racks to your partition.

> **INFO**
>
> The hardware that we currently support to be placed inside a partition is described in the hardware document.

> **INFO**
>
> How large you can grow your partitions and how the network topology inside a partition looks like is described in the networking document.

The metal-stack has microservices running on the leaf switches in a partition. For this reason, your leaf switches are required to run a Linux distribution that you have full access to. Additionally, there are a servers not added to the pool of user-allocatable machines, which are instead required for running metal-stack and we call them *management servers*. We also call the entirety of switches inside a partition the *switch plane*.

The microservices running inside a partition are:

- **metal-hammer** (runs on a server when not allocated by user, often referred to as *discovery image*) An initrd, which is booted up in PXE mode, preparing and registering a machine. When a user allocates a machine, the metal-hammer will install the target operating system on this machine and kexec into the new operating system kernel.
- **metal-core** (runs on leaf switches) Dynamically configures the leaf switch from information provided by the metal-api. It also proxies requests from the metal-hammer to the metal-api including publishment of machine lifecycle events and machine registration requests.
- **pixiecore** (preferably runs on management servers, forked by metal-stack) Provides the capability of PXE booting servers in the PXE boot network.
- **metal-bmc** (runs on management servers) Reports the ip addresses that are leased to ipmi devices together with their machine uuids to the metal-api. This provides machine discovery in the partition machines and keeps all IPMI interface access data up-to-date. Also forwards metal-console requests to the actual machine, allowing user access to the machine's serial console. Furthermore it processes firmware updates and power on/off, led on/off, boot order changes.

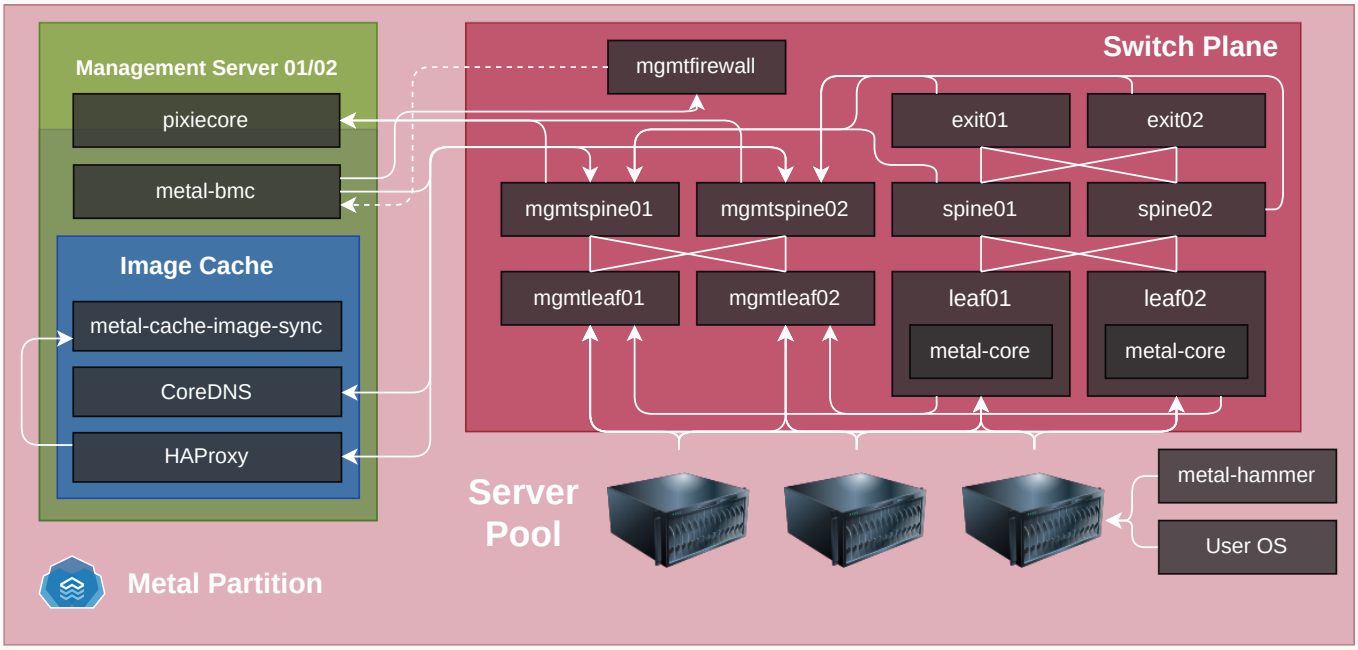


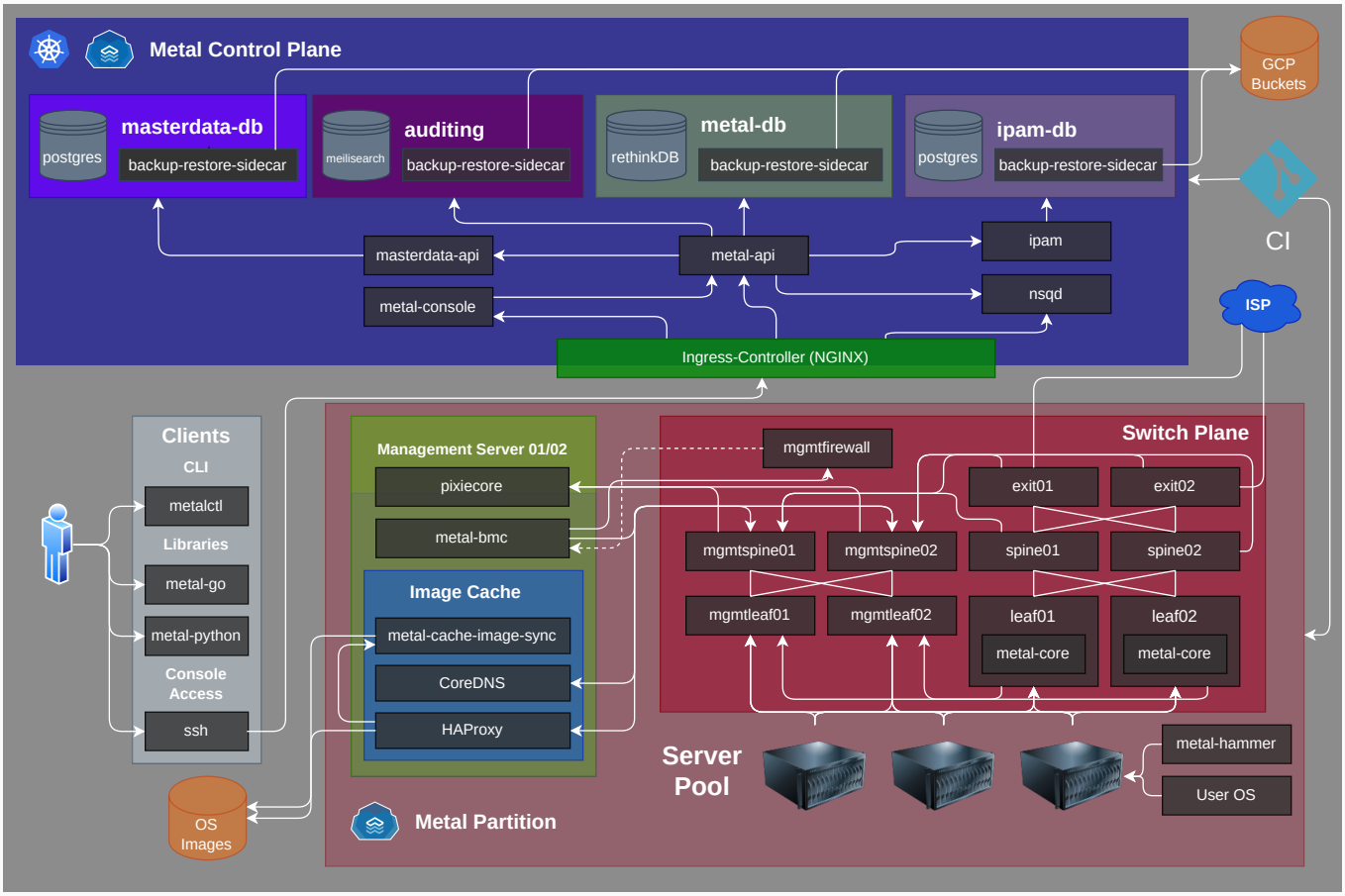
Figure 2: Simplified illustration of services running inside a partition.

Some notes on this picture:

- This figure is slightly simplified. The switch plane consists of spine switches, exit routers, management firewalls and a bastion router with more software components deployed on these entities. Please refer to the networking document to see the full overview over the switch plane.
- The image-cache is an optional component consisting of multiple services to allow caching images from the public image store inside a partition. This brings increased download performance on machine allocation and increases independence of a partition on the internet connection.

Complete View

The following figure shows several partitions connected to a single metal control plane. Of course, it is also possible to have multiple metal control planes, which can be useful for staging.



> Figure 3: Reduced view on the communication between the metal control plane and multiple partitions.

Some notes on this picture:

- By design, a partition only has very few ports open for incoming-connections from the internet. This contributes to a smaller attack surface and higher security of your infrastructure.
- With the help of NSQ, it is not required to have connections from the metal control plane to the metal-core. The metal-core instances register at the message bus and can then consume partition-specific topics, e.g. when a machine deletion gets issued by a user.

Machine Provisioning Sequence

The following sequence diagram illustrates some of the main principles of the machine provisioning lifecycle.

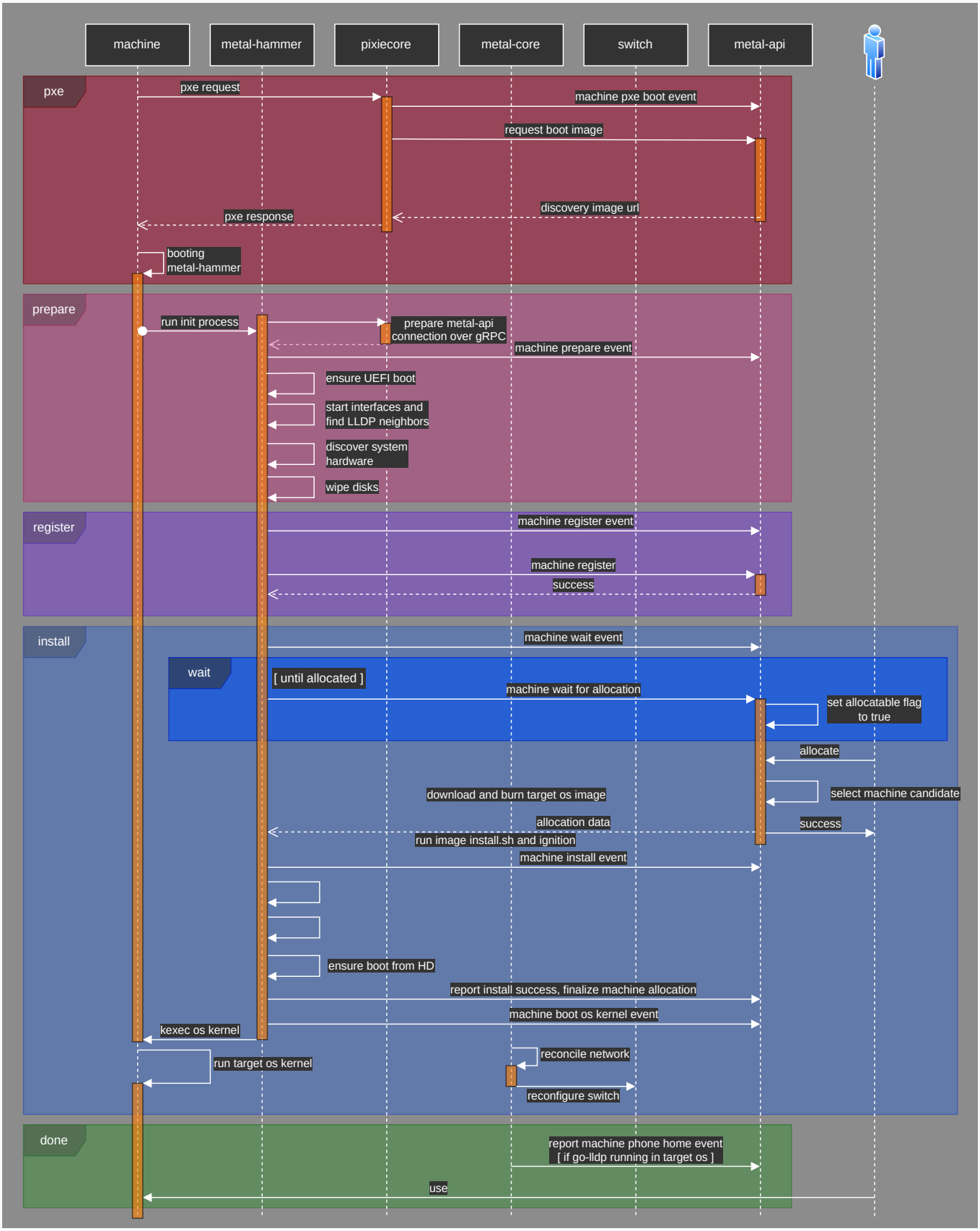
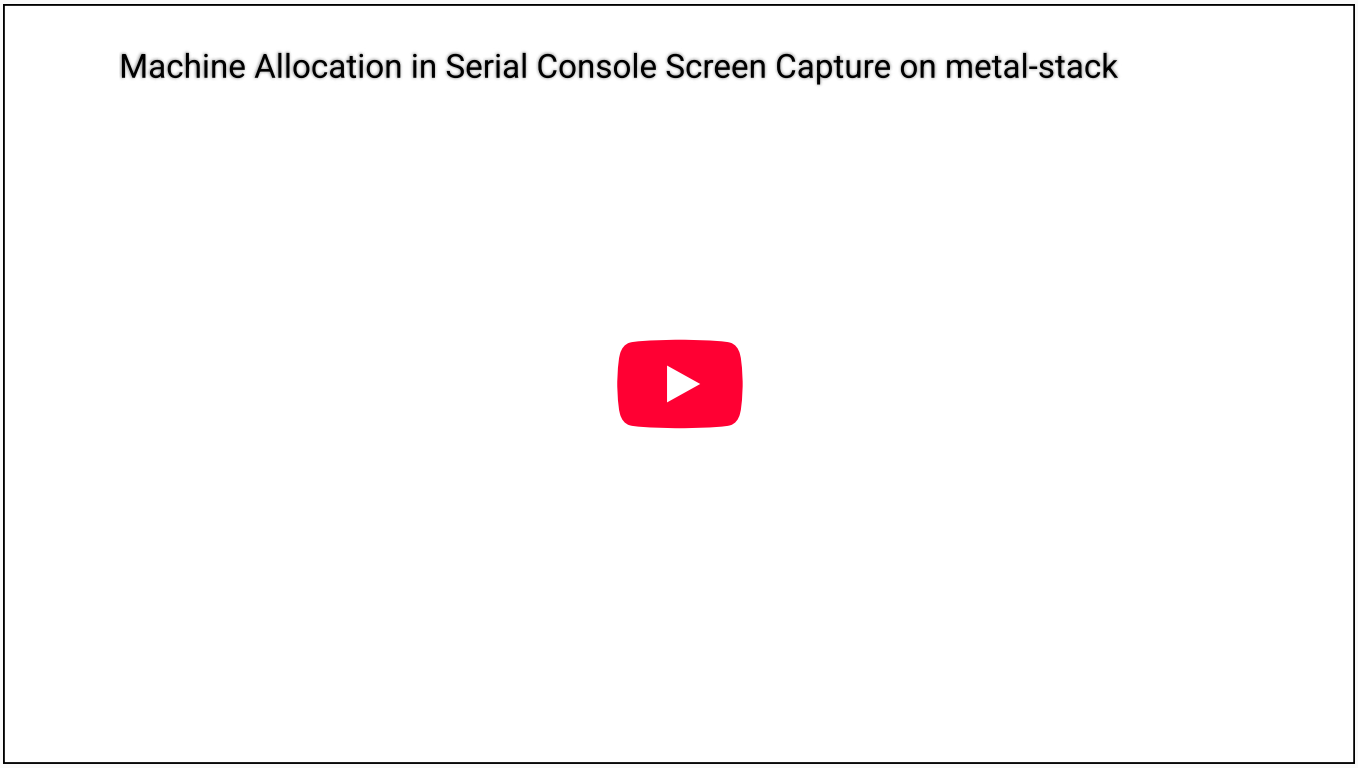


Figure 4: Sequence diagram of the machine provisioning sequence.

Here is a video showing a screen capture of a machine's serial console while running the metal-hammer in "wait mode". Then, a user allocates the machine and the metal-hammer installs the target operating system and the machine boots into the new operating system kernel via the kexec system call.



Offline Resilience

It is possible to use metal-stack without any external network dependencies by integrating your own DNS and NTP configuration into the stack. This feature is great for workloads requiring strong independence and reliability. Even in case of an internet connection failure, your infrastructure remains operational. Existing machines do not encounter any downtime as well as new machines can be provisioned. All you need to have in place is a DNS and NTP server configured and accessible for metal-stack.

NTP servers need to be configured on the pixiecore and the metal-hammer microservices. This can be achieved by providing a list of NTP servers with the following Ansible variable through metal-roles:

```
pixiecore_metal_hammer_ntp_servers: []
```

In the background, the pixiecore is taking the NTP servers and passing it via the `MetalConfig` to the metal-hammer. When booting bare-metal servers, the metal-hammer needs to configure NTP servers. It recognises the ones from the `MetalConfig` and configures itself accordingly. If no NTP servers are passed along, the following standard servers are used:

- 0.de.pool.ntp.org
- 1.de.pool.ntp.org
- 2.de.pool.ntp.org

Moreover, machine and firewall images need to be configured with your custom DNS and NTP servers. The customisation can be made via the fields `ntp_servers` an `dns_servers` and specifying a list of servers in the creation request for the machine or firewall.

Within a partition default values for DNS and NTP servers can be configured. They are applied to all machines and firewalls within this partition, but can be replaced by specifying different ones inside the machine allocation request.

Thus, for creating a partition as well as a machine or a firewall, the flags `dnsservers` and `ntpservers` can be provided within the `metalctl` command.

In order to be fully offline resilient, make sure to check out `metal-image-cache-sync`. This component provides copies of `metal-images`, `metal-kernel` and `metal-hammer`.

This feature is related to MEP14.

User Management

At the moment, metal-stack can more or less be seen as a low-level API that does not scope access based on projects and tenants. Fine-grained access control with full multi-tenancy support is actively worked on in MEP4.

Until then projects and tenants can be created, but have no effect on access control.

Default Users

The current system provides three default users with their corresponding roles:

- **Metal-Admin** is an **Admin** can perform all actions.
- **Metal-Edit** has the **Edit** role and may create, edit and delete most resources.
- **Metal-Viewer** is a **Viewer** and may only view resources and may access machines.

Each of these users have a corresponding HMAC token, which can be used to authenticate against the API. The tokens do not expire and can be used as long as the service is running. Be cautious with who you share the tokens with.

OIDC

Currently the only way to act as a different user than the default ones, is by using OIDC authentication. Here the OIDC provider decides which role the user has.

Role Mapping

The following table shows which role is required to access the endpoints of the various services at a high level. Only the minimum role required to access the group of endpoints is shown. For the more in-depth documentation of all endpoints, head over to the API documentation.

Service	Group of Endpoints	Minimum Role
audit-service	Reading audit traces	Viewer
filesystem-service	Reading filesystem layouts	Viewer
	Managing filesystem layouts	Admin
firewall-service	Reading firewalls	Viewer
	Allocating firewalls	Editor

Service	Group of Endpoints	Minimum Role
firmware-service	All endpoints	Admin
image-service	Reading images	Viewer
	Managing images	Admin
ip-service	Reading IPs	Viewer
	Managing IPs	Editor
machine-service	Reading machines and issues	Viewer
	Managing machines and issues	Editor
	IPMI operations	Editor
	Updating, deleting machines	Admin
	Updating firmware	Admin
network-service	Reading networks	Viewer
	Allocating and freeing networks	Editor
	Managing networks	Admin
partition-service	Reading partitions	Viewer
	Managing partitions	Admin
project-service	Reading projects	Viewer
	Managing projects	Admin
size-service	Reading sizes	Viewer
	Managing reservations	Editor
	Managing sizes	Admin
sizeimageconstraint-service	Reading size image constraints	Viewer
	Managing size image constraints	Admin
switch-service	Reading switches	Viewer
	Managing switches	Admin

Service	Group of Endpoints	Minimum Role
tenant-service	Reading tenants	Viewer
	Managing tenants	Admin
user-service	Getting user information	Viewer
vpn-service	Getting VPN auth key	Admin

Networking

We spent a lot of time on trying to provide state-of-the-art networking in the data center. This document describes the requirements, ideas and implementation details of the network topology that hosts the metal-stack.

The document is separated into three main sections describing the constraints, theoretical ideas and implementation details.

Requirements

Finding the requirements for this greenfield project was kicked off with a handful of design parameters that included:

- Investigation of the idea of a **layer-3 based infrastructure** to overcome the drawbacks of traditional layer-2 architectures.
- Application of a routing technology that involves a single stand-alone protocol **BGP** for operational simplicity.
- Utilization of the overlay virtual network technology **EVPN** to support cost-effective scaling, efficient network information exchange and a manageable amount of administration effort.
- Applying the routing topology on top of a completely new physical infrastructure that is designed as a CLOS network topology.

Evaluation of those parameters led to more specific requirements:

- Physical Wiring:
  - The data center is made of a leaf-spine CLOS topology containing:
    - leaf switches
    - spine switches
    - exit switches
    - management server
    - management switch
    - tenant servers
    - tenant firewalls.
  - Bare metal servers are dual-attached to leaf switches. The bare metal servers either become tenant servers or firewalls for a group of tenant servers.
  - All network switches are connected to a management switch. A management server provides access to this management network.
- Network Operation Characteristics:
  - IPv4 based network.
  - No IPv6 deployment.
  - Utilization of external BGP.
  - Numbered BGP only for peerings at exit switches with third parties (Internet Service Provider).
  - Overall BGP unnumbered.
  - 4-byte private ASN instead of default 2-byte ASN for BGP.
  - Network operation relies on SONiC Linux.
  - Bleeding edge Routing-to-the-Host/EVPN-to-the-Host with ordinary Linux distributions.
  - Layer-3 routing using BGP and VXLAN/EVPN.

- Every VTEP acts as a layer-3 gateway and does routing. Routing is done on both the ingress and the egress VTEP (aka distributed symmetric routing).
- Tenant isolation is realized with VRF.
- Internet Access is implemented with route leak on the firewall servers and during the PXE-Process with route leak on the exit switches.
- MTU 9216 is used for VXLAN-facing interfaces, otherwise MTU 9000 is used.

Furthermore, requirements such as *operational simplicity* and *network stability* that *a small group of people can effectively support* have been identified being a primary focus for building metal-stack.

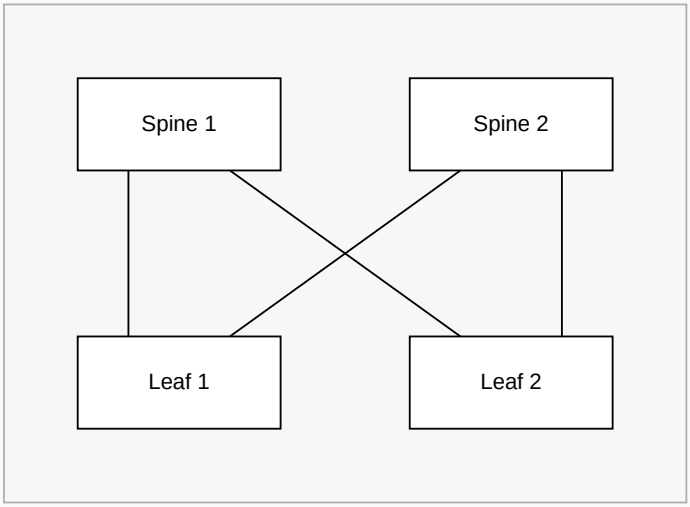
Concept

The theoretical concept targets the aforementioned requirements. New technologies have been evaluated to apply the best solutions. The process was heavily inspired by the work of Dinesh G. Dutt regarding BGP (bgp-ebook), EVPN (evpn-ebook) and the his 2019 work "Cloud Native Data Center Networking" (O'Reilly), which teaches some interesting basics.

External BGP together with network overlay concepts as EVPN can address the essential demands. These revolutionary concepts are part of the next evolutionary step in data center design. It overcomes common issues of traditional layer 2 architectures (e.g. VLAN limitations, network visibility for operations, firewall requirements) by introducing a layer 3 based network topology.

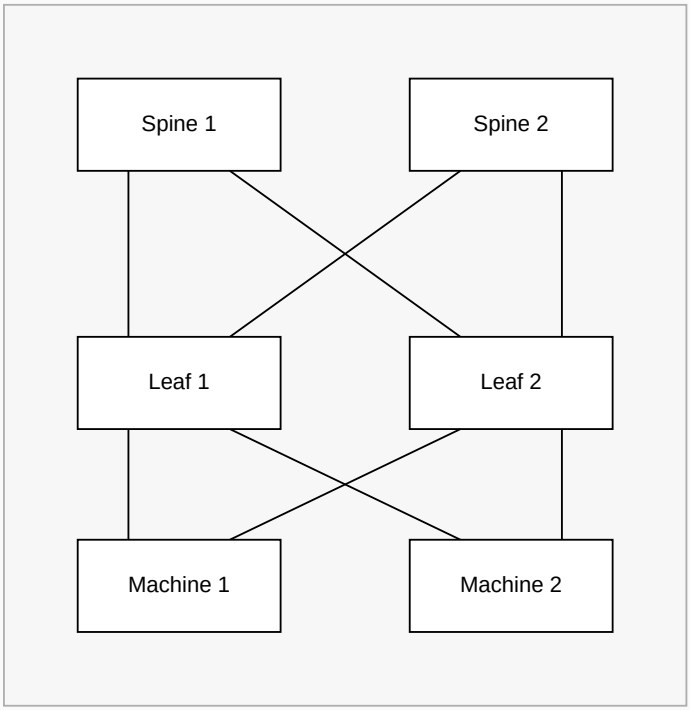
CLOS

A CLOS topology is named after the pioneer Charles Clos (short: **CLOS**) who first formalized this approach. CLOS defines a multistage network topology that is used today to improve performance and resilience while enabling a cost effective scalability. A CLOS topology comprises network switches aggregated into spine and leaf layers. Each leaf switch (short: **leaf**) is connected to all spine switches (short: **spine**) but there is no direct leaf-to-leaf or spine-to-spine connection (See: picture 1).



> Picture 1: Fragment of CLOS to show leaf-spine layer.

This data center network architecture, based on a leaf-spine architecture, is also know as "two-tier" CLOS topology.



> Picture 2: Fragment to show a 3-stage, 2-layer CLOS topology.

Tenant servers are dual-attached to the leaf layer in order to have redundancy and load balancing capability (Picture 2). The set of leaves, spine switches and tenant servers define stages. From top down each server is reachable with 3 hops (spine → leaf → server). This is why that CLOS design is called a 3-stage CLOS. Consistent latency throughout the data center are an outcome of this design.

It is not only important to have a scalable and resilient infrastructure but also to support planning and operation teams. Visibility within the network is of significant meaning for them. Consequently layer-3 routing in favor of layer-2 bridging provides this kind of tooling.

BGP

For routing the **Border Gateway Protocol (BGP)**, more specific: External BGP was selected. Extensive testing and operational experiences have shown that External BGP is well suited as a stand-alone routing protocol (see: RFC7938).

Not all tenant servers are connected to the same leaf. Instead they can be distributed among any of the leaves of the data center. To not let this detail restrict the intra-tenant communication it is required to interconnect those layer-2 domains. In the context of BGP there is a concept of overlay networking with VXLAN/ EVPN that was evaluated to satisfy the needs of the metal-stack.

BGP Unnumbered

In BGP traditionally each BGP peer-facing interface requires a separate IPv4 address. This consumes a lot of IP addresses. RFC 5549 defines the BGP unnumbered standard. It allows to use interface's IPv6 link local address (LLA) to set up a BGP session with a peer. With BGP unnumbered the IPv6 LLA of the remote is automatically discovered via Router Advertisement (RA) protocol. Important: This does not (!) mean that IPv6 must be deployed in the network. BGP uses RFC 5549 to encode IPv4 routes as reachable over IPv6 next-hop using the LLA. Having unnumbered interfaces does not mean no IPv4 address may be in place. It is a good practice to configure an IP address to the never failing and always present local loopback interface (lo). This lo address is reachable over BGP from other peers because the RFC 5549 standard provides an encoding scheme to allow a router to advertise IPv4 routes with an IPv6 next-hop. BGP unnumbered also has an advantage from security perspective. It removes IPv4 and global IPv6 addresses from router interfaces, thus reducing the attack vector.

To sum it up:

- BGP unnumbered uses IPv6 next-hops to announce IPv4 routes.
- There is no IPv6 deployment in the network required.
- IPv6 just has to be enabled on the BGP peers to provide LLA and RA.

In External BGP, ASN is how BGP peers know each other.

ASN Numbering

Within the data center each BGP router is identified by a private autonomous system number (ASN). This ASN is used for internal communication. The default is to have 2-byte ASN. To avoid having to find workarounds in case the ASN address space is exhausted, a 4-byte ASN (see RFC 6793) that supports up to 95 million private ASNs (4200000000–4294967294, see RFC 6996) is used from the beginning.

ASN numbering in a CLOS topology should follow a model to avoid routing problems (path hunting) due to it's redundant nature. Within a a two-tier CLOS topology the following ASN numbering model is suggested to solve path hunting problems:

- Leaves have unique ASN
- Spines share an ASN
- Exit switches share an ASN

A illustrated example of the background of this architecture decision can be inspected in the chapter "BGP's ASN Numbering Scheme" ("BGP'S PATH HUNTING PROBLEM") of the previously mentioned "Cloud Native Data Center Networking" book.

To summarize that, one can say: Since all nodes receive or know the physical connection status of all other nodes in the network, the nodes potentially have routing information that they do not know whether they still have up to date, since it takes some time before they are fully distributed in the network. Routes to nodes may actually no longer exist (because not a single link to the node, but the node itself has failed) or the path may have changed. To determine how and whether a particular node can be reached, a path search must therefore be carried out at all its communication partners or BGP routers. Essentially, the sharing of ASNs reduces the transmission of incorrect or outdated path information (this reduces path transmissions and calculations and thus saves resources).

Address-Families

As stated, BGP is a multi-protocol routing protocol. Since it is planned to use IPv4 and overlay networks using EVPN/VXLAN several address-families have to be activated for the BGP sessions to use:

- IPv4 unicast address-family
- L2 EVPN address-family

EVPN

Ethernet VPN (EVPN, see RFC 7432) is an overlay virtual network that connects layer-2 segments over layer-3 infrastructure. EVPN is an answer to common problems of entire layer-2 data centers.

The necessity of EVPN

Challenges such as large failure domains, spanning tree complexities, difficult troubleshooting and scaling issues are addressed by EVPN:

- **administration**: less routers are involved in configuration (with VLAN every switch on routing-paths needs VLAN awareness). The configuration is less error prone due to the nature of EVPN and the good support in FRR.
- **scaling**: EVPN overcomes scaling issues with traditional VLANs (max. 4094 VLANs).
- **cost-effectiveness**: EVPN is an overlay virtual network. Not every switch on the routing path needs EVPN awareness. This enables the use of standard routers (in contrast to traditional VLAN); e.g.: spine switches act only as EVPN information replicator and do not need to have knowledge of specific virtual networks.
- **efficiency**: EVPN information is exclusively exchanged via BGP (Multiprotocol BGP, see RFC 4760). Only a single eBGP session is needed to advertise layer-2 reachability. No other protocols beneath BGP are involved and flood traffic is reduced to a minimum (no "flood-and-learn", no BUM traffic).

Virtual routing permits multiple network paths without the need of multiple switches. Hence the servers are logically isolated by assigning their networks to dedicated virtual routers using virtual routing and forwarding (short, **VRF**, see Linux Virtual Routing and Forwarding and SONiC VRF support).

The operation of EVPN

EVPN (technology) is based on BGP as control plane protocol (underlay) and VXLAN as data plane protocol (overlay).

As EVPN is an overlay network, only the VXLAN Tunnel End Points (VTEPs) must be configured. In the case of two-tier CLOS networks leaf switches are tunnel endpoints.

As described earlier, a dedicated VRF is used for each new tenant. VRF enables true multi-tenancy/isolation for routing tables. This is why the same ip-addresses or -networks can be used for tenants with different meanings without collisions or conflicts.

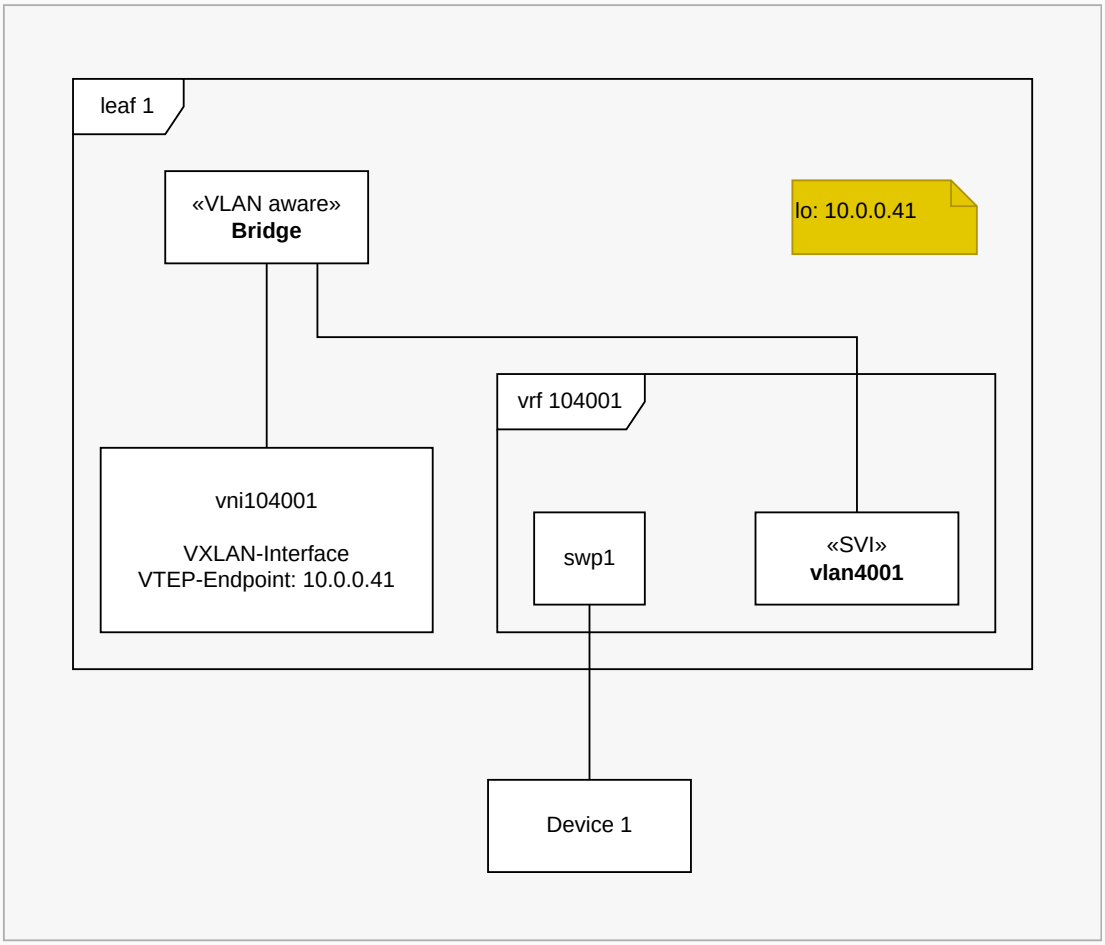
In EVPN routing is assumed to occur in the context of a VRF. VRF enables true multitenancy/isolatation for routing tables. Therewith, VRF is the first step for EVPN configuration and there is a 1:1 relationship between tenant and VRF.

To enable layer-2 connectivity, we need a special interface to route between layer-2 networks. This interface is called Switched VLAN Interface (SVI). The SVI is realized with a VLAN. It is part of a VRF (layer-3).

The VTEP configuration requires the setup of a VXLAN interface. A VLAN aware bridge interconnects the VXLAN interface and the SVI.

Required resources to establish the EVPN control plane:

- VRF: because routing happens in the context of this interface.
- SVI: because remote host routes for symmetric routing are installed over this interface.
- VLAN-aware bridge: because router MAC addresses of remote VTEPs are installed over this interface.
- VXLAN Interface / VXLAN Tunnel Endpoint: because the VRF to layer-3 VNI mapping has to be consistent across all VTEPs)



> Picture 3: Required interfaces on the switch to wire up the vrf to swp 1 connectivity with a given vxlan

Integrated routing and bridging (IRB) is the most complex part of EVPN. You could choose between centralized or distributed routing, and between asymmetrical (routing on ingress) or symmetrical (routing on ingress and egress) routing. We expect a lot of traffic within the data center itself which implies the need to avoid zigzag routing. This is why we go with distributed routing model. Further it is recommended to use the symmetric model since it makes the cut in most cases and has advantages in scalability (see "EVPN in the Data Center", Dinesh G. Dutt).

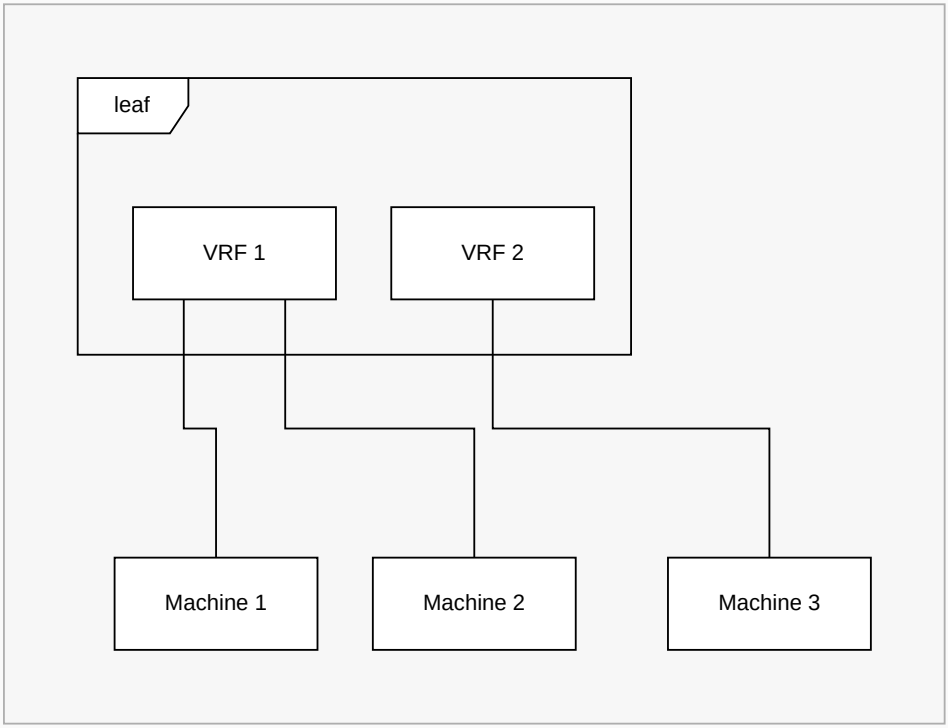
MTU

In a layer-3 network it is important to associate each interface with a proper Maximum Transmission Unit (MTU) to avoid fragmentation of IP packets. Typical modern networks do not fragment IP packets and the introduction of VXLAN adds another additional header to the packets that must not exceed the MTU. If the MTU is exceeded, VXLAN might just fail without error. This already represents a difficult-to-diagnose connectivity issue that has to be avoided.

It is common practice to set the MTU for VXLAN facing interfaces (e.g. inter-switch links) to a value of `9216` to compensate the additional VXLAN overhead and an MTU of `9000` as a default to other interfaces (e.g. server facing ports). The common MTU of `1500` is not sufficient for traffic inside a data center!

VRF

Routing is needed for communication between VXLAN tunnels or between a VXLAN tunnel and an external networks. VXLAN routing supports layer-3 multi-tenancy. All routing occurs in the context of a VRF. There is a 1:1 relation of a VRF to a tenant. Picture 3 illustrates this. Servers A and B belong to the same vrf VRF1. Server C is enslaved into VRF2. There is no communication possible between members of VRF1 and those of VRF2.



> Picture 4: Illustration of two distinct routing tables of VRF1 (enslaved: servers A and B) and VRF2 (enslaved: server C)

To leaverage the potential and power of BGP, VRF, EVPN/VXLAN without a vendor lock-in the implementation relies on hardware that is supported by open network operating system: SONiC.

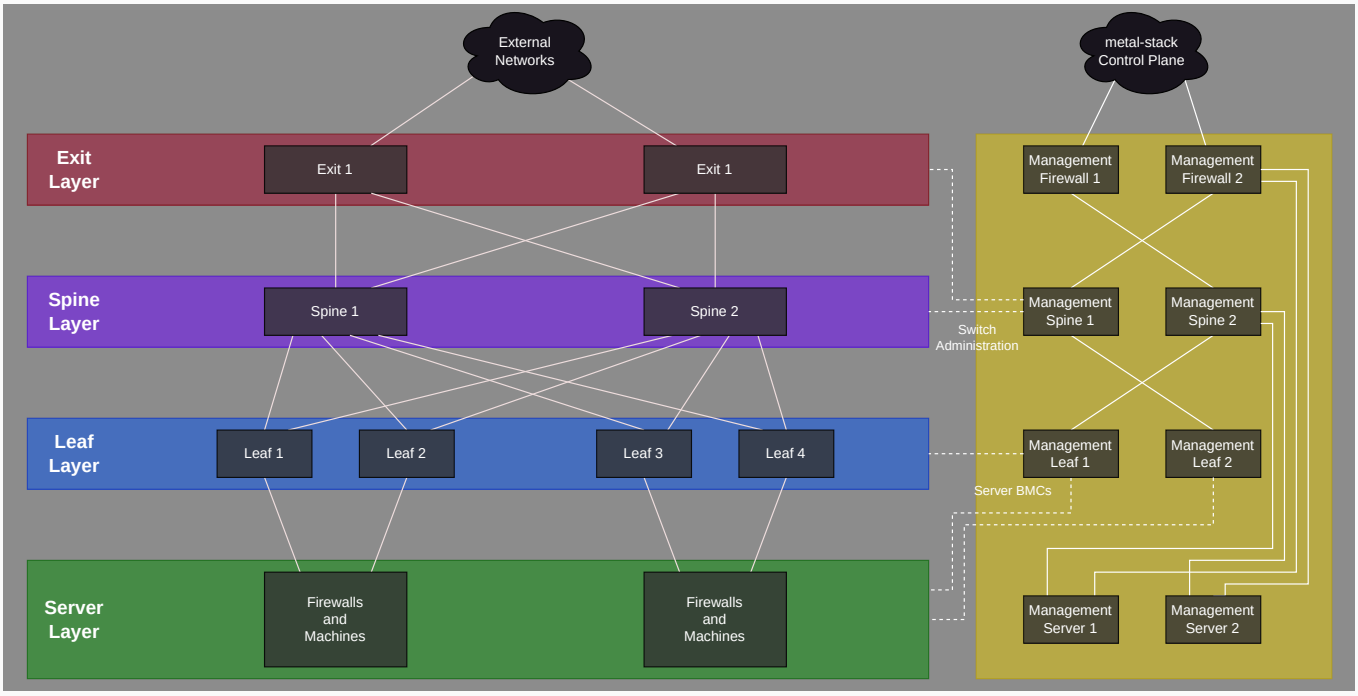
Implementation

Implementation of the network operation requires the data center infrastructure to be in place. To implement a functional meaning for the parts of the CLOS network, all members must be wired accordingly.

Physical Wiring

Reference: See the CLOS overview picture

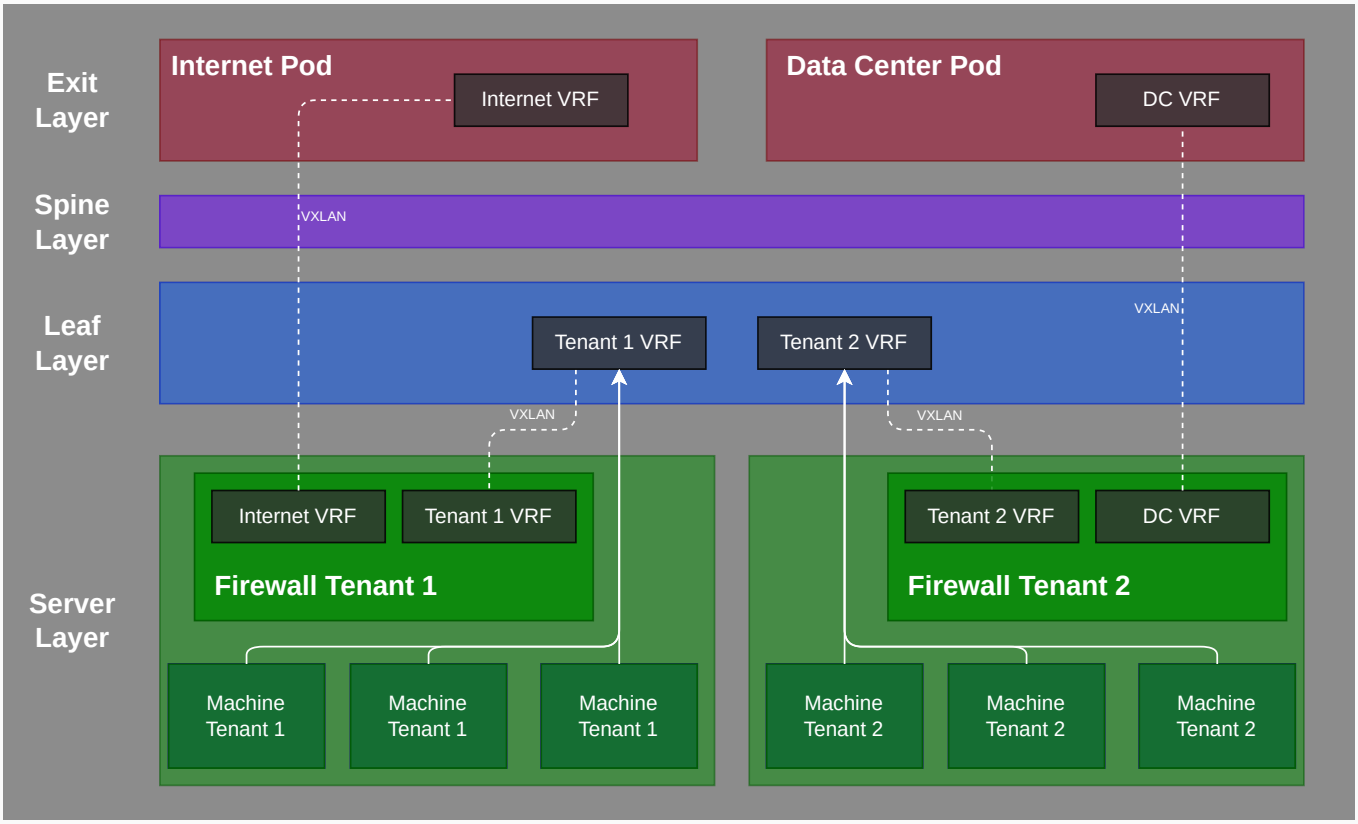
Name	Wiring
Tenant server (aka Machine)	Bare metal server that is associated to a tenant. Dual-connected to leafs.
Tenant firewall	Bare metal server that is associated to a tenant. Dual-connected to leafs.
Leaf	Network Switch that interconnects tenant servers and firewalls. Connected to spines.
Spine	Network switch that interconnects leafs and exit switches.
Exit	Network switch that connects to spines and interconnects to external networks.
Management Server	Jump-host to access all network switches within the CLOS topology for administrative purpose.
Management Switch	Connected to the management port of each of the network switches.



Picture 5: This illustration shows an example of a suitable physical wiring inside a metal-stack partition.

Tenant servers are organized into a layer called projects. In case those tenant servers require access to or from external networks, a new tenant server to function as a firewall is created. Leaf and spine switches form the fundament of the CLOS network to facilitate redundancy, resilience and scalability. Exit switches establish connectivity to or from external networks. Management Switch and Management Server are mandatory parts that build a management network to access the network switches for administration.

To operate the CLOS topology, software defined configuration to enable BGP, VRF, EVPN and VXLAN must be set up.



> Picture 6: This illustration shows the VRF tenant separation and VRF termination happening on the firewall for the tenant VRF and external network VRFs.

Network Operating Systems

SONiC as the network operating system will be installed on all network switches (leaves, spines, exit switches) within the CLOS topology. SONiC cannot be installed on bare metal servers that require BGP/EVPN but does not have a switching silicon.

Components without a switching silicon are:

- tenant servers
- tenant firewalls
- management server

There exist two paradigms to use BGP and/or VXLAN/EVPN on non switching bare metal servers: **BGP-to-the-host** and **EVPN-to-the-host**. Both describe a setup of Free Range Routing Framework (see

frrouting.org) and its configuration. FRR seamlessly integrates with the native Linux IP networking stacks.

Starting with an explanation of the tenant server's BGP-to-the-Host helps to get an insight into the setup of the CLOS network from a bottom-up perspective.

Tenant Servers: BGP-to-the-Host

Tenant servers are dual-connected to leaf switches. To communicate with other servers or reach out to external networks they must join a BGP session with each of the leaf switches. Thus, it is required to bring BGP to those hosts (aka BGP-to-the-Host). Each tenant server becomes a BGP router (aka BGP speaker).

BGP-to-the-Host is established by installing and configuring FRR. The required FRR configuration for tenant servers is limited to a basic setup to peer with BGP next-hops:

```
# /etc/network/interfaces

auto lo
iface lo inet static
  address 10.0.0.1/32

auto lan0
iface lan0 inet6 auto
  mtu 9000

auto lan1
iface lan1 inet6 auto
  mtu 9000
```

> Listing 1: Network interfaces of a tenant server.

Listing 1 shows the local interfaces configuration. lan0 and lan1 connect to the leaves. As described, there is no IPv4 address assigned to them (BGP unnumbered). The local loopback has an IPv4 address assigned that is announced by BGP.

The required BGP configuration:

```
# /etc/frr/frr.conf

frr version 7.0
frr defaults datacenter
log syslog debugging
service integrated-vtysh-config
!
interface lan0
  ipv6 nd ra-interval 6
  no ipv6 nd suppress-ra
!
interface lan1
  ipv6 nd ra-interval 6
  no ipv6 nd suppress-ra
!
router bgp 4200000001
  bgp router-id 10.0.0.1
```

```
  bgp bestpath as-path multipath-relax
  neighbor TOR peer-group
  neighbor TOR remote-as external
  neighbor TOR timers 1 3
  neighbor lan0 interface peer-group TOR
  neighbor lan1 interface peer-group TOR
  neighbor LOCAL peer-group
  neighbor LOCAL remote-as internal
  neighbor LOCAL timers 1 3
  neighbor LOCAL route-map local-in in
  bgp listen range 10.244.0.0/16 peer-group LOCAL
  address-family ipv4 unicast
    redistribute connected
    neighbor TOR route-map only-self-out out
  exit-address-family
!
bgp as-path access-list SELF permit ^$
!
route-map local-in permit 10
  set weight 32768
!
route-map only-self-out permit 10
 match as-path SELF
!
route-map only-self-out deny 99
!
```

> Listing 2: FRR configuration of a tenant server.

The frr configuration in Listing 2 starts with `frr defaults datacenter`. This is a marker that enables compile-time provided settings that e.g. set specific values for BGP session timers. This is followed by a directive to state that instead of several configuration files for different purposes a single *frr.conf* file is used: `service integrated-vtysh-config`. The two interface specific blocks starting with `interface ...` enable the RA mechanism that is required for BGP unnumbered peer discovery. There is a global BGP instance configuration `router bgp 4200000001` that sets the private ASN. The BGP router configuration contains a setup that identifies the BGP speaker `bgp router-id 10.0.0.1`. This router id should be unique. It is a good practice to assign the local loopback IPv4 as router-id. To apply the same configuration to several interfaces a peer group named `TOR` is defined via `neighbor TOR peer-group`. `remote-as external` activates external BGP for this peer group. To have a fast convergence, limits of default timers are reduced by `timer 1 3` section. The two BGP-peer-facing interfaces are enslaved into the peer-group to inherit the peer-group's setup. Activation of IPv4 unicast protocol is completed with `address-family ipv4 unicast`. To prevent a tenant server from announcing other paths than `lo` interface a route-map `only-self-out` is defined. This route map is activated within the ipv4 address family: `neighbor TOR route-map only-self-out out`.

Application of the route map `only-self-out` enables to announce only local ip(s). This is to avoid that a tenant server announces paths to other servers (prevents unwanted traffic). To achieve this:

- the route-map named `only-self-out` permits only matches against an access list named `SELF`
- access list `SELF` permits only empty path announcements

- the path of the tenant server itself has no ASN. It is always empty (see line `*> 10.0.0.2/32 0.0.0.0 0 32768 ?`):

```
root@machine:~# vtysh -c 'show bgp ipv4 unicast'
BGP table version is 7, local router ID is 10.0.0.2, vrf id 0
Default local pref 100, local AS 4200000002
Status codes:  s suppressed, d damped, h history, * valid, > best, = multipath,
               i internal, r RIB-failure, S Stale, R Removed
Nexthop codes: @NNN nexthop's vrf id, < announce-nh-self
Origin codes:  i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
*= 0.0.0.0/0        lan1                                   0 4200000012 4200000040 i
*>                  lan0                                   0 4200000011 4200000040 i
*= 10.0.0.1/32      lan1                                   0 4200000012 4200000001 ?
*>                  lan0                                   0 4200000011 4200000001 ?
*> 10.0.0.2/32      0.0.0.0                  0         32768 ?
*= 10.0.0.78/32     lan1                                   0 4200000012 4200000001 ?
*>                  lan0                                   0 4200000011 4200000001 ?

Displayed  4 routes and 7 total paths
```

  That is why only the self ip (loopback ip) is announced.

To allow for peering between FRR and other routing daemons on a tenant server a `listen range` is specified to accept iBGP sessions on the network `10.244.0.0/16`. Therewith it gets possible that pods / containers like metal-lb with IPs of this range may peer with FRR.

This is the only place where we use iBGP in our topology. For local peering this has the advantage, that we don't need an additional ASN that has to be handled / pruned in the AS-path of routes. Routes coming from other routing daemons look as if they are configured on the tenant server's lo interface from the viewpoint of the leaves. iBGP routes are differently handled than eBGP routes in BGPs best path algorithm. Generally BGP has the rule to prefer eBGP routes over iBGP routes (see 'eBGP over iBGP'). BGP adds automatically an weight based on the route type. To overcome this issue, we set the weight of iBGP routes to the same weight that eBGP routes have, namely 32768 (`set weight 32768`). Without this configuration we will only get a single route to the IPs announced via iBGP. So this setting is essential for HA/failover!

Statistics of the established BGP session can be viewed locally from the tenant server via: `sudo vtysh -c 'show bgp ipv4 unicast'`

To establish this BGP session a BGP setup is required on the leaves as well.

Leaf Setup

Every leaf switch is connected to every spine switch. Tenant servers can be distributed within the data center and thus be connected to different leaves. Routing for tenant servers is isolated in unique VRFs. These constraints imply several configuration requirements for the leaf and spine switches:

- leaves define tenant VRFs
- leaves terminate VXLAN tunnels (aka "VXLAN tunnel endpoint" = VTEP)

The leaf setup requires the definition of a tenant VRF that enslaves the tenant server facing interfaces:

```
# /etc/network/interfaces

# [...]

iface vrf3981
    vrf-table auto

iface swp1
    mtu 9000
    post-up sysctl -w net.ipv6.conf.swp1.disable_ipv6=0
    vrf vrf3981

# [...]
```

> Listing 3: Fragment that shows swp1 being member of vrf vrf3981.

There is a VRF definition `iface vrf3981` to create a distinct routing table and a section `vrf vrf3981` that enslaves swp1 (connects the tenant server) into the VRF. Those host facing ports are also called `edge ports`.

Additional to the VRF definition the leaf must be configured to provide and connect a VXLAN interface to establish a VXLAN tunnel. This network virtualization begins at the leaves. Therefore, the leaves are also called Network Virtualization Edges (NVEs). The leaves encapsulate and decapsulate VXLAN packets.

```
# /etc/network/interfaces

# [...]

iface bridge
    bridge-ports vni3981
    bridge-vids 1001
    bridge-vlan-aware yes

iface vlan1001
    mtu 9000
    vlan-id 1001
    vlan-raw-device bridge
    vrf vrf3981

iface vni3981
    mtu 9000
    bridge-access 1001
    bridge-arp-nd-suppress on
    bridge-learning off
    mstpctl-bpduguard yes
    mstpctl-portbpdufilter yes
    vxlan-id 3981
    vxlan-local-tunnelip 10.0.0.11

# [...]
```

> Listing 4: Fragment that shows VXLAN setup for vrf vrf3981.

All routing happens in the context of the tenant VRF. To send and receive packets of a VRF, several interface are in place.

A bridge is used to attach VXLAN interface `bridge-ports vni3981` and map its local VLAN to a VNI. Router MAC addresses of remote VTEPs are installed over this interface.

The Routed VLAN Interface or Switched Virtual Interface (SVI) `iface vlan1001` is configured corresponding to the per-tenant VXLAN interface. It is attached to the tenant VRF. Remote host routes are installed over this SVI. The `vlan-raw-device bridge` is used to associate the SVI with the VLAN aware bridge. For a packet received from a locally attached host the SVI interface corresponding to the VLAN determines the VRF `vrf vrf3981`.

The VXLAN interface `iface vni3981` defines a tunnel address that is used for the VXLAN tunnel header `vlxan-local-tunnelip 10.0.0.11`. This VTEP IP address is typically the loopback device address of the switch. When EVPN is provisioned, data plane MAC learning for VXLAN interfaces must be disabled because the purpose of EVPN is to exchange MACs between VTEPs in the control plane: `bridge-learning off`. EVPN is responsible for installing remote MACs. `bridge-arp-nd-suppress` suppresses ARP flooding over VXLAN tunnels. Instead, a local proxy handles ARP requests received from locally attached hosts for remote hosts. ARP suppression is the implementation for IPv4; ND suppression is the implementation for IPv6. It is recommended to enable ARP suppression on all VXLAN interfaces. Bridge Protocol Data Unit (BPDU) are not transmitted over VXLAN interfaces. So as a good practice bpduguard and pbdufilter are enabled with `mstpctl-bpduguard yes` and `mstpctl-portbpdufilter yes`. These settings filter BPDU and guard the spanning tree topology from unauthorized switches affecting the forwarding path. `vxlan-id 3981` specifies the VXLAN Network Identifier (VNI). The type of VNI can either be layer-2 (L2) or layer-3 (L3). This is an implicit thing. A VNI is a L3 VNI (L3VNI) when a mapping exists that maps the VNI to a VRF (configured in `/etc/frr/frr.conf`) otherwise it is a L2 VNI (L2VNI).

```
# /etc/frr/frr.conf
# [...]
vrf vrf3981
 vni 3981
 exit-vrf
#[...]
router bgp 4200000011
# [...]
 address-family ipv4 unicast
  redistribute connected route-map LOOPBACKS
 # [...]
 address-family l2vpn evpn
  neighbor FABRIC activate
  advertise-all-vni
 exit-address-family
# [...]
router bgp 4200000011 vrf vrf3981
 # [...]
 address-family ipv4 unicast
  redistribute connected
  neighbor MACHINE maximum-prefix 100
 exit-address-family
 !
 address-family l2vpn evpn
  advertise ipv4 unicast
```

```
 exit-address-family

# [...]
route-map LOOPBACKS permit 10
 match interface lo
# [...]
```

Listing 5: Leaf FRR configuration.

Listing 5 shows the required FRR configuration of the BGP control plane. Only content not discussed so far is explained. The section `vrf vrf3981` contains the mapping from layer-3 VNI to VRF. This is required to be able to install EVPN IP prefix routes (type-5 routes) into the routing table. Further the file contains a global BGP instance `router bgp 4200000011` definition. A new setting `redistribute connected route-map LOOPBACKS` is in place to filter the redistribution of routes that are not matching the local loopback interface. The route-map is defined with `route-map LOOPBACKS permit 10`. With the configuration line `address-family l2vpn evpn`, the EVPN address family is enabled between BGP neighbours. `advertise-all-vni` makes the switch a VTEP configures it in such a way, that all locally configured VNIs should be advertised by the BGP control plane.

The second BGP instance configuration is specific to the tenant VRF `router bgp 4200000011 vrf vrf3981`. This VRF BGP instance configures the l2vpn evpn address family with `advertise ipv4 unicast` to announce IP prefixes in BGP's routing information base (RIB). This is required to apply learned routes to the routing tables of connected hosts. The Maximum-Prefix feature is useful to avoid that a router receives more routes than the router memory can take. The maximum number of prefixes a tenant server is allowed to announce is limited to `100` with: `neighbor MACHINE maximum-prefix 100`.

Spine setup

On the spine switches the setup is quite simple. `/etc/network/interfaces` contains the loopback interface definition to support BGP unnumbered and listings for connected switch ports to provide proper MTUs (Listing 6). I.e. swp1 is configured with an MTU of 9216 as it is a VXLAN-facing interface.

```
# /etc/network/interfaces
# [...]
iface swp1
    mtu 9216
```

Listing 6: Fragment of spine interface configuration.

The spines are important to forward EVPN routes and transport VXLAN packets between the VTEPs. They are not configured as VTEPs. The FRR configuration only contains the already known global BGP instance configuration `router bgp 4200000020` plus the activation of the l2vpn evpn address family `address-family l2vpn evpn` to enable EVPN type-5 route forwarding (Listing 7).

```
hostname spine01
username admin nopassword
!
# [...]
```

```
interface swp1
 ipv6 nd ra-interval 6
 no ipv6 nd suppress-ra
!
# [...]
!
router bgp 4200000020
 # [...]
!
 address-family l2vpn evpn
  neighbor FABRIC activate
 exit-address-family
!
# [...]
```

> Listing 7: Fragment of spine FRR configuration to show the activated L2VPN EVPN address-family.

Tenant Firewalls: EVPN-to-the-Host

In case a tenant server needs to reach out to external networks as the Internet, a tenant firewall is provisioned. The firewall is a bare metal server without a switching silicon. Thus, there is no installation of SONiC. FRR provides the BGP / EVPN functionality known as `EVPN-to-the-host`. The firewall is configured as a VTEP and applies `dynamic route-leaking` to install routes of an foreign VRF. The set of routes that are leaked are restricted with route-maps.

As Listing 8 shows, the firewall is configured with VXLAN interfaces as known from the leaf setup. Additionally, a VXLAN setup for VRF `vrfInternet` is added to provide Internet access. vrfInternet contains a route to the Internet that will be leaked into the tenant VRF.

Traffic that originates from the tenant network `10.0.0.0/22` will be masqueraded before leaving the interface `vlanInternet` towards the internet.

```
# /etc/network/interfaces
# [...]
iface bridge
# [...]
iface vlan1001
# [...]
iface vni3981
# [...]
iface vrf3981
# [...]
iface vlanInternet
    mtu 9000
    vlan-id 4009
    vlan-raw-device bridge
    vrf vrfInternet
    address 185.1.2.3
    post-up iptables -t nat -A POSTROUTING -s 10.0.0.0/22 -o vlanInternet -j MASQUERADE
    pre-down iptables -t nat -D POSTROUTING -s 10.0.0.0/22 -o vlanInternet -j MASQUERADE

iface vniInternet
    mtu 9000
    bridge-access 4009
    mstpctl-bpduguard yes
```

```
    mstpctl-portbpdufilter yes
    vxlan-id 104009
    vxlan-local-tunnelip 10.0.0.40

iface vrfInternet
    mtu 9000
    vrf-table auto
```

Listing 8: Interfaces configuration of firewall to show the VTEP interface configuration.

To install a default route into the routing table of tenant VRF vrf3981 a dynamic route leak is established for it (`import vrf vrfInternet`). With the help of a route-map `import vrf route-map vrf3981-import-map` only the default route will be leaked:

```
root@firewall01:~# vtysh -c 'show ip route vrf vrf3981'
# [...]
VRF vrf3981:
S>* 0.0.0.0/0 [1/0] is directly connected, vrfInternet(vrf vrfInternet), 03:19:26
B>* 10.0.0.1/32 [20/0] via 10.0.0.12, vlan1001 onlink, 02:34:48
  *                    via 10.0.0.11, vlan1001 onlink, 02:34:48
B>* 10.0.0.2/32 [20/0] via 10.0.0.12, vlan1001 onlink, 02:34:49
  *                    via 10.0.0.11, vlan1001 onlink, 02:34:49
```

To receive responses from vrfInternet in vrf3981 a route is leaked into vrfInternet as well (`import vrf vrf3981`) restricted with the route-map `vrfInternet-import-map` that allows leaking of the tenant routes as well as internet prefixes used on worker nodes of the tenant. To limit the prefixes that are announced from the firewall within the global BGP instance a route-map `only-self-out` is defined and applied within the ipv4 and l2vpn evpn address family. Together with the definition of an as path access list `bgp as-path access-list` it avoids the announcement of prefixes to non VRF BGP peers.

```
# /etc/frr/frr.conf
!
vrf vrf3981
 vni 3981
!
vrf vrfInternet
 vni 104009
!
# [...]
!
router bgp 4200000040
 # [...]
 !
 address-family ipv4 unicast
  # [...]
  neighbor FABRIC route-map only-self-out out
 exit-address-family
 !
!
router bgp 4200000040 vrf vrf3981
 # [...]
 address-family ipv4 unicast
  redistribute connected
  import vrf vrfInternet
```

```
  import vrf route-map vrf3981-import-map
 # [...]
 address-family l2vpn evpn
  advertise ipv4 unicast
 # [...]
router bgp 4200000040 vrf vrfInternet
 # [...]
 address-family ipv4 unicast
  redistribute connected
  import vrf vrf3981
  import vrf route-map vrfInternet-import-map
 # [...]
 address-family l2vpn evpn
  advertise ipv4 unicast
 # [...]
 bgp as-path access-list SELF permit ^$
!
route-map only-self-out permit 10
 match as-path SELF
!
route-map only-self-out deny 99
!
route-map LOOPBACKS permit 10
 match interface lo
!
ip prefix-list vrf3981-import-prefixes seq 100 permit 0.0.0.0/0
!
route-map vrf3981-import-map permit 10
 match ip address prefix-list vrf3981-import-prefixes
!
route-map vrf3981-import-map deny 99
!
ip prefix-list vrfInternet-import-prefixes seq 100 permit 10.0.0.0/22 le 32
ip prefix-list vrfInternet-import-prefixes seq 101 permit 185.1.2.0/24 le 32
ip prefix-list vrfInternet-import-prefixes seq 102 permit 185.27.0.0/27 le 32
!
route-map vrfInternet-import-map permit 10
 match ip address prefix-list vrfInternet-import-prefixes
!
route-map vrfInternet-import-map deny 99
!
line vty
!
```

> Listing 9: FRR configuration of a tenant firewall to show route leak and prefix announcement filtering.

Exit Switch

Traffic to external networks is routed via the firewalls to the exit switch. The exit switch, as an exception, connects to the Internet Service Provider using numbered BGP. Numbered BGP implies to assign IPv4 addresses to network interfaces (See Listing 10, swp1). Interface swp1 is enslaved into `vrf vrfInternet` to include the port that is connected to the ISP within the VRF that is expected to contain a way into the Internet. The exit switch is configured to be a VTEP to terminate traffic coming from the firewall VRF `vrfInternet`.

```
# /etc/network/interfaces
# [...]
iface swp1
    mtu 9000
    vrf vrfInternet
    address 172.100.0.2/30
# [...]
iface vlan4000
    mtu 9000
    address 10.0.0.71/24
    vlan-id 4000
    vlan-raw-device bridge
# [...]
iface vlanInternet
# [...]
iface vniInternet
# [...]
iface vrfInternet
# [...]
```

> Listing 10: Fragment of interfaces configuration of exit switch.

The configuration of FRR is equivalent to the previously discussed ones. It contains a global BGP instance configuration that enables IPv4 unicast and l2vpn evpn address families. The vrfInternet BGP instance defines `neighbor 172.100.0.1 peer-group INTERNET` to use "old style BGP" transit network.

```
# [..]
vrf vrfInternet
 vni 104009
!
# [...]
router bgp 4200000031
 bgp router-id 10.0.0.31
 neighbor FABRIC peer-group
 neighbor FABRIC remote-as external
 neighbor FABRIC timers 1 3
 # [...]
 !
 address-family ipv4 unicast
  neighbor FABRIC activate
  redistribute connected route-map LOOPBACKS
 exit-address-family
 !
 address-family l2vpn evpn
  neighbor FABRIC activate
  advertise-all-vni
 exit-address-family
!
router bgp 4200000031 vrf vrfInternet
 bgp router-id 10.0.0.31
 bgp bestpath as-path multipath-relax
 neighbor INTERNET peer-group
 neighbor INTERNET remote-as external
 neighbor INTERNET timers 1 3
 neighbor 172.100.0.1 peer-group INTERNET
 !
 address-family ipv4 unicast
  neighbor INTERNET route-map PREPEND-PATH-TO-DISFAVOR-IN in
```

```
  neighbor INTERNET route-map PREPEND-PATH-TO-DISFAVOR-OUT out
 exit-address-family

 !
 address-family l2vpn evpn
  advertise ipv4 unicast
 exit-address-family
!
route-map LOOPBACKS permit 10
 match interface lo
!
route-map PREPEND-PATH-TO-DISFAVOR-IN permit 10
 set as-path prepend last-as 2
!
route-map PREPEND-PATH-TO-DISFAVOR-OUT permit 10
 set as-path prepend last-as 2
!
vrf mgmt
 ip route 10.0.0.0/24 10.0.0.71 nexthop-vrf default
 exit-vrf
!
ip route 0.0.0.0/0 192.168.0.254 nexthop-vrf mgmt
!
line vty
!
```

> Listing 11: Fragment of FRR configuration on exit switch to give an example for numbered BGP and route leak.

In addition to the standard BGP setup the exit switches have configured `static route leak` to support internet access during PXE. There is one route leak from `default` VRF into the `mgmt` VRF defined with: `ip route 0.0.0.0/0 192.168.0.254 nexthop-vrf mgmt` and another one from `mgmt` VRF into the `default` VRF: `ip route 10.0.0.0/24 10.0.0.71 nexthop-vrf default`. The first one adds a default route into the `default` VRF and the second one routes traffic destined to the PXE network back from `mgmt` VRF into the `default` VRF.

To reach out into external networks each of the exit nodes joins a BGP session with a distinct external router. There is a different latency to each of these routers. To favor routes of exit nodes connected with lower latency over exit nodes with higher latency two route maps `PREPEND-PATH-TO-DISFAVOR-IN` and `PREPEND-PATH-TO-DISFAVOR-OUT` are added to high latency exit nodes. These route maps apply actions to prolong the path of the incoming and outgoing routes. Because of this path extension BGP will calculate a lower weight for these paths and favors paths via other exit nodes. It is important to know that within an address family only one route map (the last) will be applied. To apply more than one actions within a route-map the required entries can be applied to a single route-map.

PXE Boot Mode

Before a bare metal server can act as tenant server or tenant firewall, it has to be provisioned. Within the Metal domain, this provisioning mode is called "PXE Mode" since it is based on Preboot eXecution Environment (PXE). PXE uses protocols like DHCP. This requires all bare metal servers that need

provisioning to be located in a layer-2 domain where DHCP is available. This domain is a VLAN `vlan4000`. A DHCP server for PXE Mode is installed on the exit switches to work in this specific VLAN.

```
# /etc/default/isc-dhcp-server
INTERFACES="vlan4000"
```

> Listing 13: DHCP server configuration of exit switches.

As shown in listing 13, the PXE DHCP server is located on the exit switches and enforced to bind to interface `vlan4000`. This represents a layer-2 separation that allows only DHCP clients in the same VLAN to request IP addresses. Only unprovisionned bare metal servers are configured to be member of this VLAN. Thus unwanted or accidental provisionning is impossible.

To provide `vlan4000` on the leaves (that face the bare metal servers) the exit and leaf switches are configured as VTEPs and share an interface configuration that contains the required interfaces (Listing 13). Since no EVPN routing is in place `vni104000` is configured as an L2 VNI (there is no mapping for this VNI in `/etc/frr/frr.conf`).

```
# /etc/network/interfaces
# [...]
iface bridge
    bridge-ports vni104000 [...]
    bridge-vids 4000 [...]
    bridge-vlan-aware yes

iface vlan4000
# [...]

iface vni104000
# [...]
```

> Listing 13: Interfaces configuration on exit and leaf switches to show DHCP/PXE related fragments.

On the leaf switches the bare metal server facing ports are configured as VLAN access ports to carry the traffic for only the PXE VLAN `vlan4000` (listing 14)to separate unprovisioned from other bare metal servers.

```
# /etc/network/interfaces
# [...]
auto swp1
iface swp1
    mtu 9000
    bridge-access 4000
# [...]
```

> Listing 14: VLAN access setup for bare metal server facing ports on leaves.

Once a bare metal server is provisioned it is deconfigured from PXE VLAN `vlan4000` to avoid accidental or unwanted provisioning.

During provisioning bare metal servers get internet access via the management network of the exit switches. This is because the exit switches are announced as DHCP gateway to the DHCP clients.

Management Network

To manage network switches beside the out-of-band system console access a further management access is required. For this purpose the concept of **Management VRF** is applied. The Management VRF is a subset of VRF. It provides a separation between out-of-band management network and the in-band data plane network by introducing another routing table **mgmt**. SONiC supports eth0 to be used as the management interface.

To enable and use the Management VRF all switches have to be connected via their eth0 interface to a management-switch. The management switch is connected to a management server. All access is established from within the management server. Logins to the switch are set into the Management VRF context once the Management VRF is enabled.

DRAFT PAGE

This page is a draft. It will only be visible in dev and be excluded from the production build.

Firewalls

(fire-walling in metal-stack, firewall-controller and headscale integration)

Expose Services with Tailscale

This guide is a recommendation on how to access services from anywhere if you are evaluating the metalstack on-prem starter without a public IP-address.

These steps will guide you through the process quickly. For a deeper dive, or if you want to use alternative setups, the Tailscale articles are also linked.

What are Tailscale and Tailnets?

Tailscale is a Canadian company that offers a virtual private network solution based on WireGuard.

Instead of relying on centralised VPN servers to route all traffic, Tailscale creates a mesh VPN called Tailnet. It creates encrypted peer-to-peer connections between network subscribers. This approach is claimed to improve throughput and stability while reducing latency.

Tailscale's solution is composed of components that are mostly open source. Find more information on their open source sstatement and their GitHub repository.

Setup an Account and Clients

Please begin by following these steps to use an authentication provider to create the first user account for your network.

The first step is to install Tailscale clients on the devices from which you wish to access the Kubernetes services.

Should you require to extend an invitation to additional users, this can be facilitated by navigating to the "Users" tab on the Admin Console.

Setup the Operator

Labels

First, we will establish tags to categorise our services. Please open the 'Access controls' tab, where you will find a text editor containing all the Access Control settings in JSON format.

Uncomment the `tagOwners` section and add the following tags:

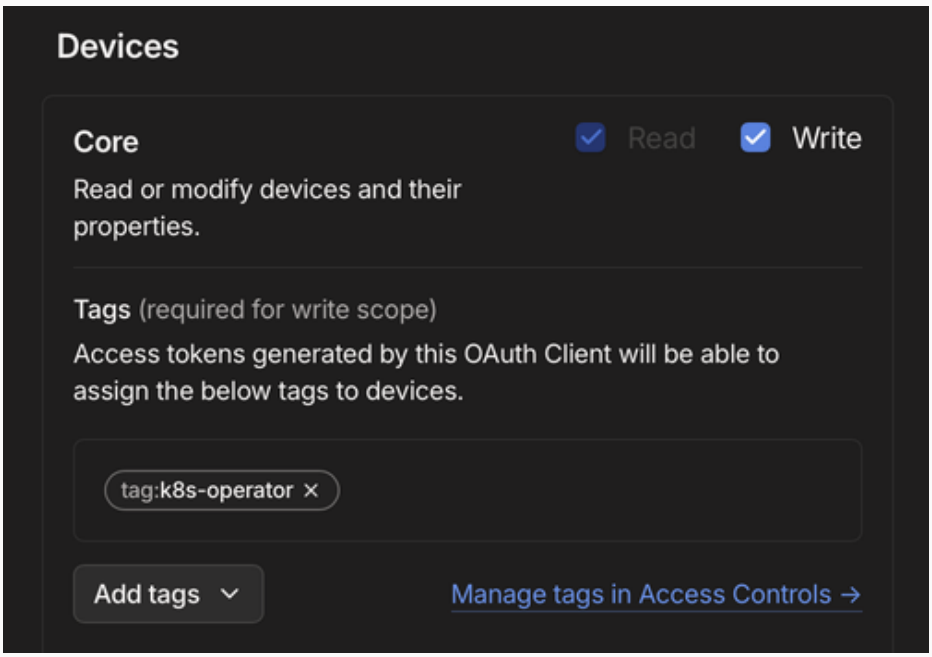
```
"tagOwners": {
   "tag:k8s-operator": [],
```

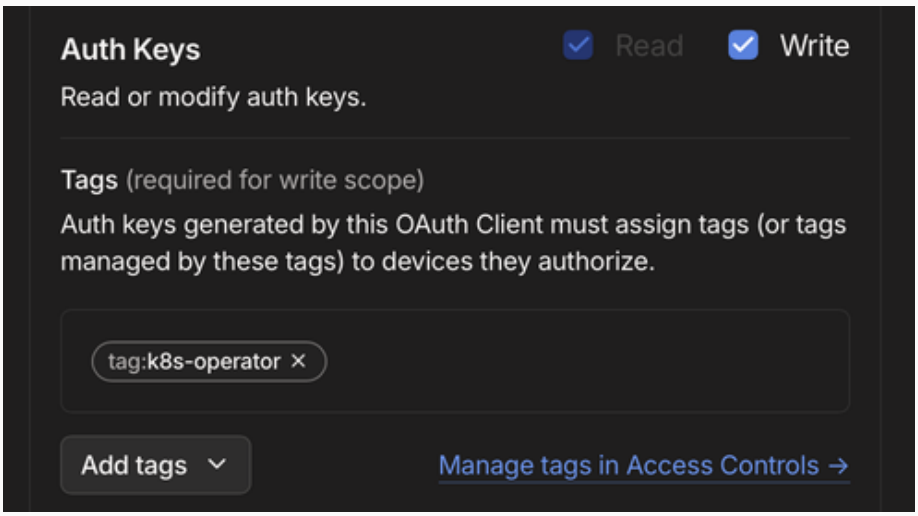
```
    "tag:k8s": ["tag:k8s-operator"],
}
```

The operator will use the `k8s-operator`-tag. Devices with this tag are now configured as owner for devices with the `k8s`-tag, which will be used for our services.

Create OAuth-Client Credentials

In the "Settings" tab at "OAuth clients", generate a new OAuth client. Set write permissions for "Devices - Core" and "Keys - Auth Keys". Select the `k8s-operator` tag for both.





Therefore, a device that has been assigned the label `k8s-operator` will have the capability to register additional devices with the `k8s` tag.

When you click "create", you get a client-id and client-secret, that you will need to setup the operator.

Setup Operator with helm

The most common and practical way is to use a Helm chart to setup the operator. Therefore, we first have to add and update the helm-repository of tailscale:

```
helm repo add tailscale https://pkgs.tailscale.com/helmcharts
helm repo update
```

Now, we can install the helm-chart in a dedicated namespace using the credentials of the OAuth client

```
helm upgrade \
  --install \
  tailscale-operator \
  tailscale/tailscale-operator \
  --namespace=tailscale \
  --create-namespace \
  --set-string oauth.clientId="<OAauth client ID>" \
  --set-string oauth.clientSecret="<OAuth client secret>" \
  --wait
```

Check on the administration console, if your operator appears on the Machines list.

Alternative ways & troubleshooting:

- Take the operator.yaml manifest from the GitHub repository and make your adjustments to use Static manifests with kubectl
- If the operator does not show op in the Machines list, use the guide for Troubleshooting the Kubernetes operator

Expose Services on the Tailnet

There are three ways to allow traffic to your pods from the tailnet. We can setup a dedicated Service or annotate an existing one. For more routing options, use an Ingress object. For detailed configuration options, review the article about Expose a Kubernetes cluster workload to your tailnet (cluster ingress)

Add a Load Balancer Service

The installed operator is looking for Service objects with the `spec.type` of `LoadBalancer` and the `spec.loadBalancerClass` of `tailscale`.

```
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  ports:
    - name: https
      port: 443
      targetPort: 443
```

```
  type: LoadBalancer
  loadBalancerClass: tailscale
```

Annotate an existing Service

Edit the Service and add the annotation `tailscale.com/expose` with the value "true":

```
---
apiVersion: v1
kind: Service
metadata:
  annotations:
    tailscale.com/expose: "true"
  name: nginx
spec:
  ...
```

Note that "true" is quoted because annotation values are strings, and an unquoted true will be incorrectly interpreted as a boolean.

Use an Ingress

To enable path-based routing, use an Ingress resource. Ingress routes only use TLS over HTTPS. To make this work, you have to enable the `MagicDNS` and `HTTPS` options in the DNS-Tab on your Administration Console. This enables helpful features:

- Magic DNS automatically register your services with subdomains in your tailnet
- HTTPS enables the provisioning of certificates for devices

To set the Ingress up, just refer to `tailscale` as the `ingressClassName`:

```
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx
spec:
  ingressClassName: tailscale
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
  ...
```

Please consider, that currently only paths with `pathType: prefix` are supported currently

Gardener

Gardener is an open source project for orchestrated Kubernetes cluster provisioning. It supports many different cloud providers, metal-stack being one of them. Using the Gardener project, metal-stack can act as a machine provider for Kubernetes worker nodes.

The idea behind the Gardener project is to start with a dedicated set of Kubernetes clusters (this can be a single cluster, too), which are used to host Kubernetes control planes for new Kubernetes clusters. The new Kubernetes control planes reside in dedicated namespaces of the initial clusters ("Kubernetes in Kubernetes" or "underlay / overlay Kubernetes"). Where initial clusters come from is the subject of a larger debate, with suggestions made in a later section of this document.

Gardener's architecture is designed for multi-tenant environments, with a strong distinction between the operator and the end users. In Gardener, Kubernetes control planes for different tenants may reside in the same operator cluster. This approach makes it very suitable for being used with bare metal because it allows taking full advantage of the server resources. Another implication is that end users do not have access to their control plane components, such as the kube-apiserver or the ETCD. These are managed by the operator and in case of metal-stack even physically divided from the end user's workload.

Gardener allocates machines from a cloud provider and automatically deploys a kubelet to those nodes, which then joins the appropriate control plane. Operators can also nest clusters so that newly provisioned clusters can be used to spin up more clusters, leading to nearly infinite scalability (also known as "kubeception" model).

Terminology

We would like to explain the most important Gardener terms. The terminology used in the Gardener project has many similarities to the architecture of Kubernetes. Additional information can also be found in the official glossary.

Garden Cluster

The Garden Cluster is a Kubernetes cluster that runs the Gardener Control Plane.

The control plane components introduce dedicated Kubernetes API resources for provisioning new Kubernetes clusters with the Gardener. It also takes care of the validation for many of those Gardener API resources and also reconciling some of them. The components are the following:

- Gardener API Server
- Gardener Controller Manager
- Gardener Scheduler
- Gardener Admission Controller

The control plane components can be deployed in the Garden Cluster through the Gardener Operator.

The Garden cluster can also be used as seed cluster.

Virtual Garden

A recommended way to deploy the Gardener is running a "virtual cluster" inside the Garden cluster. It is basically a Kubernetes control plane without any worker nodes, providing the Kubernetes API in an own ETCD. Its purpose is to store all Gardener resources (such that they reside inside a dedicated ETCD) and provide an individual update lifecycle from the Garden Cluster. End users can have access to own project namespaces in the virtual garden, too.

The virtual garden consists of the following components:

- garden kube-apiserver
- etcd
- kube-controller-manager

More details about the virtual garden can be found in the description of `gardener-operator`.

Seeds and Soils

A seed cluster is a cluster in which an agent component called the `Gardenlet` is running. The gardenlet is connected to the Gardener Control Plane and is responsible for orchestrating the provisioning of new clusters inside the seed cluster. The control plane components for the new clusters run as pods in the seed cluster.

A seed cluster can also be called a soil if the Gardenlet has been manually deployed by the operator and not by the Gardener. Clusters created on the soil can be turned into seed clusters by the operator using a Gardener resource called `ManagedSeed`. This resource causes Gardener to automatically deploy the Gardenlet to the new cluster, such that the resulting cluster is not called a soil.

Shoot

Every Kubernetes cluster that is fully provisioned and managed by Gardener is called a `Shoot` cluster. It consists of the shoot control plane running on the seed cluster and worker nodes running the actual workload.

Gardener Integration Components

During the provisioning flow of a cluster, Gardener emits resources that are expected to be reconciled by controllers of a cloud provider. This section briefly describes the controllers implemented by metal-stack to allow the creation of a Kubernetes cluster on metal-stack infrastructure.

If you want to learn how to deploy metal-stack with Gardener, please check out the corresponding deployment-guide section.

gardener-extension-provider-metal

The gardener-extension-provider-metal contains of a set of webhooks and controllers for reconciling cloud provider specific resources of `type: Metal`, which created by Gardener during the cluster provisioning flow.

Primarily, its purpose is to reconcile `Infrastructure`, `ControlPlane`, and `Worker` resources.

The project also introduces an own API (`ProviderConfiguration` resources) and consists of an admission-controller to validate them. This admission controller should be deployed in the Gardener control plane cluster.

os-metal-extension

Due to the reason metal-stack initially used ignition to provision operating system images (today, cloud-init is supported as well) there is an implementation of a controller that translates the generic `OperatingSystemConfig` format of Gardener into ignition userdata. It can be found on Github in the os-metal-extension repository.

machine-controller-manager-provider-metal

Worker nodes are managed through Gardener's machine-controller-manager (MCM). The MCM allows out-of-tree provider implementation via sidecar, which is what we implemented in the machine-controller-manager-provider-metal repository.

Initial Cluster Setup

Before creating the `garden cluster`, a base K8s cluster needs to be in place. Some suggestions for the initial K8s cluster are:

- GCP/GKE
- metalstack.cloud

Initial Cluster on GCP

- A GCP account needs to be in place.
- The Ansible gcp-auth role can be used for authenticating against GCP.
- The Ansible gcp-create role can be used for creating a GKE cluster.

Suggestions for default values are:

- `gcp_machine_type`: e2-standard-8
- `gcp_autoscaling_min_nodes`: 1
- `gcp_autoscaling_max_nodes`: 3

Initial Cluster on metalstack.cloud

- A Kubernetes cluster can be created on metalstack.cloud via UI, CLI or Terraform.

metal-stack Setup

> **Attention:** Bootstrapping a metal-stack partition is out of scope and need to be done before focusing on the relationship between metal-stack and Gardener. This guide assumes a metal-stack partition (servers, switches, network, ...) is already in place.

Start by deploying:

- `ingress-nginx-controller`
- `cert-manager`

This guide assumes, that metal-stack gets deployed on the same initial cluster as Gardener. On the initial cluster, the metal-stack control plane need to be deployed. This can be done as described in the metal-stack documentation.

Garden Cluster Setup

After setting up the initial K8s cluster and metal-stack, Gardener can be deployed with the Gardener Ansible role.

This deploys the following components:

- virtual garden
- Gardener control plane components
- soil cluster
- managed seed cluster (into the metal-stack partition)

In summary, this results in the following:

- `Garden cluster` created in the initial cluster
- `soil cluster` created in the initial cluster. This will be the `initial seed` used for spinning up `shooted seeds` in the metal-stack partition
- `shooted seed` inside the metal-stack partition, where all `shoots` are derived from

Cluster API

Cluster API is a Kubernetes project that aims to simplify the management of Kubernetes clusters. It provides a declarative way to create, configure, and manage clusters using Kubernetes-style APIs.

We provide the Cluster API provider for metal-stack (CAPMS) infrastructure provider that allows the declaration of Kubernetes clusters.

> **"EARLY DEVELOPMENT STAGE"**
>
> This project is currently under heavy development and is not advised to be used in production any time soon. Please use our stack on top of Gardener for production workloads.

See the cluster-api-provider-metal-stack documentation for more in-depth information.

metal Cloud Controller Manager

CCM stands for cloud-controller-manager and is the bridge between Kubernetes and a cloud-provider.

We implemented the cloud provider interface in the metal-ccm repository. With the help of the cloud-controller-controller we provide metal-stack-specific properties for Kubernetes clusters, e.g. load balancer configuration through MetalLB or node properties.

Firewall Controller Manager

To make the firewalls created with metal-stack easily configurable through Kubernetes resources, we add our firewall-controller to the firewall image. The controller watches special CRDs, enabling users to manage:

- nftables rules
- Intrusion-detection with suricata
- network metric collection

Please check out the guide on how to use it.

Isolated Kubernetes Clusters

Some customers have the need to run their workloads in a very restricted environment. These restrictions are driven by regulatory requirements in some industries such as finance, healthcare, energy and more. Regulatory requirements often mandate that the workload must not be exposed to the public internet, nor is capable to reach the public internet in any case.

For this purpose we implemented a possibility to start Kubernetes clusters in such a manner. This is referred to as cluster isolation.

Design Choices

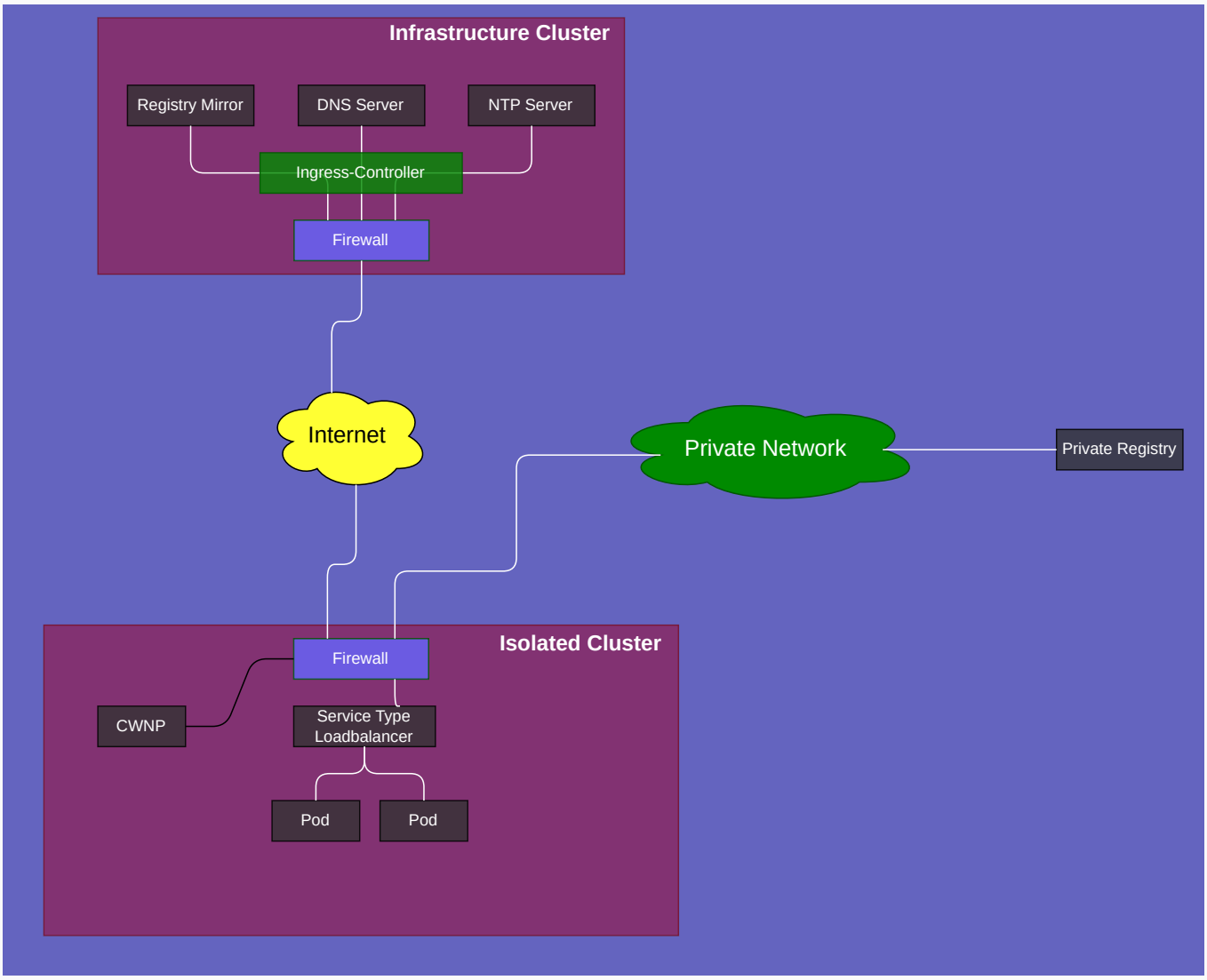
When talking about highly secure Kubernetes environments people often raise the term "Air Gapped Cluster". This would mean that no physical connection exists between the Kubernetes control plane and the Kubernetes worker nodes with the outside world. This requirement exists in extreme environments such as ships, moon bases or nuclear plants. The effort to produce this in a completely automated manner is extremely challenging.

We decided to follow a different approach which is more practical, still very secure but much simpler to implement and operate. The solution we created is called "Isolated Cluster" which means that there are still physical connections between the Kubernetes cluster, but guarded to prohibit malicious traffic. It is also not possible to enable malicious traffic by accident, e.g. if a cluster user configures network policies or load balancers to untrusted environments.

Network Design

In order to be able to restrict ingress and egress internet traffic, but still make it possible to create a working Kubernetes cluster we implemented the following network design.

- All strictly required container images are mirrored to a registry which is only accessible from the Kubernetes clusters.
- DNS and NTP servers are produced alongside the registry.
- The `containerd` configuration on every worker node is configured to pull all of the strictly required container images from this private registry mirror.
- DNS and NTP configuration is also adopted to use the DNS and NTP servers on this private environment.
- A list of networks which are allowed to reach is managed, this list reflects the networks of the cloud provider and is not modifiable by the cluster user. This list usually contains the internet prefixes of the provider and one or more RFC address ranges.



Users are advised to attach an additional network to the Kubernetes cluster in order to be able to pull container images for the application workloads from private registries.

Strictly Required Container Images

In general the creation of a Kubernetes cluster requires the ability to pull container images for several applications which are necessary to make a machine a Kubernetes worker node. To mention the most important:

- Kubelet: the main controller on each worker node to manage the workload
- CNI (Container Network Interface): controller and daemon set to setup and run the container networking
- CSI (Container Storage Interface): controller and daemon set to setup and run the container storage
- CoreDNS: DNS for containers
- MetalLB: Service Type LoadBalancer Implementation
- node-exporter and metrics-server: Monitoring for the worker node
- Metal-Stack Addons: for firewall and auditing events

Flavors

With the introduction of Isolated Kubernetes Clusters, cluster users must decide upon cluster creation which type of isolation he needs for his workload. There are three different flavours available:

- Internet access `baseline`: This is the default cluster creation mode, which does not change any aspects of network and registry access.
- Internet access `forbidden`: No internet access is possible, neither ingress nor egress.
- Internet access `restricted`: No internet access is possible, neither ingress nor egress, but can be enabled by the cluster user.

Please see the detailed description of these flavors below.

Cluster Wide Network Policies CWNP

To restrict which egress traffic is allowed, Custom Resources `ClusterWideNetworkPolicy` are deployed and can be deployed by the cluster user. The set of deployed CWNPs differs between `baseline` and `forbidden`/`restricted`.

`baseline` CWNPs:

Rule Name	Destination	Purpose
allow-to-http	0.0.0.0/0	egress via http
allow-to-https	0.0.0.0/0	egress via https
allow-to-apiserver	IP of the Kubernetes API Server on the control plane	API Server communication of kubelet and other controllers
allow-to-dns	IP of the Google DNS Servers	DNS resolution from the Kubernetes worker nodes and containers
allow-to-ntp	IP of the Cloudflare NTP Servers	Time synchronization
allow-to-storage	network of the container storage	persistent volumes with the cni driver
allow-to-vpn	IP of the vpn endpoint on the control plane	allow communication from the api server to the kubelet for container logs and container exec

`forbidden` and `restricted` CWNPs:

Rule Name	Destination	Purpose
allow-to-apiserver	IP of the Kubernetes API Server on the control plane	API Server communication of kubelet and other controllers
allow-to-dns	IP of the private DNS Server	DNS resolution from the Kubernetes worker nodes and containers

Rule Name	Destination	Purpose
allow-to-ntp	IP of the private NTP Server	Time synchronization
allow-to-registry	IP of the private Registry Mirror	Pulling strictly required container images
allow-to-storage	network of the container storage	persistent volumes with the cni driver
allow-to-vpn	IP of the vpn endpoint on the control plane	allow communication from the api server to the kubelet for container logs and container exec

All of these CWNPs are managed by the gardener-extension-provider-metal, every manual modification will be reverted immediately.

Internet Access Baseline

This is the default configuration of a Kubernetes cluster, egress traffic is controlled by multiple CWNPs (`ClusterWideNetworkPolicy`), ingress traffic is possible by deploying a Service Type LoadBalancer. The cluster user can add additional CWNPs without any restrictions and is responsible for them.

Container images can be pulled from any reachable container registry. The `containerd` is not reconfigured to point to our private registry mirror.

DNS and NTP are configured to internet DNS resolvers and NTP servers.

Internet Access Forbidden

This configuration can only be achieved by creating a new Kubernetes cluster, it is not possible to modify a existing cluster (with internet access `baseline` or `restricted`) to this configuration. It is also required to specify the most recent version of Kubernetes, older versions of Kubernetes are not supported.

Every network access modification triggered by a cluster user, either by adding or modifying CWNPs or adding a Service Type LoadBalancer, is validated against the list of allowed networks.

`containerd` is configured so that all required images are pulled from the private registry mirror. This registry contains only the strictly required images, therefore no additional (workload) images can be pulled from public registries.

Egress traffic

Egress traffic is only allowed to the private registry mirror and the DNS and NTP servers. Additional CWNPs can be added to reach destinations in the internal networks if specified. If a CWNP was created which points to a destination outside of the allowed networks, the CWNP will still be present but stays in the status `ignored`.

```
> kubectl get clusterwidenetworkpolicies.metal-stack.io
NAME                 STATUS     MESSAGE
allow-to-apiserver   deployed
allow-to-dns         deployed
allow-to-ntp         deployed
allow-to-registry    deployed
allow-to-storage     deployed
allow-to-vpn         deployed
allow-to-google      ignored    ingress/egress does not match allowed networks
```

Also an event is created which describes why the CWNP was ignored:

```
> kubectl get events
5s         Warning   ForbiddenCIDR         clusterwidenetworkpolicy/allow-to-google
address:"8.8.8.8/32" is outside of the allowed network
range:"10.0.0.0/8,100.64.0.0/10,212.34.83.0/27", ignoring
```

Ingress traffic

Ingress traffic is only allowed from the internal networks if specified. To specify the address where the Service Type LoadBalancer is listening to, the cluster user must use one of his statically acquired ip addresses. Of course, this ip address is only considered if it is contained in the list of allowed networks. Then this ip address must be configured in the service:

```
apiVersion: v1
kind: Service
spec:
  type: LoadBalancer
  loadBalancerIP: 10.1.1.1 # ip from the internal network
```

By default, no ip address will be automatically selected for such clusters and the ip of the service will stay in pending mode until the ip was specified as shown above.

```
> kubectl get svc
NAME              TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)         AGE
example-service   LoadBalancer   10.244.75.171   <pending>     443:32179/TCP   4s

> kubectl get events
8s          Warning   AllocationFailed         service/example-service   Failed to allocate IP for
"default/example-service": no available IPs
3s          Warning   SyncLoadBalancerFailed   service/example-service   Error syncing load
balancer: failed to ensure load balancer: no default network for ip acquisition specified, acquire
an ip for your cluster's project and specify it directly in "spec.loadBalancerIP"
```

Internet Access Restricted

This configuration can only be achieved by creating a new Kubernetes cluster, it is not possible to modify a existing cluster (with internet access `baseline` or `forbidden`) to this configuration. It is also required to specify the most recent version of Kubernetes, older versions of Kubernetes are not supported.

The same default CWNPs are deployed and the container images are pulled from the private registry. Also DNS and NTP are configured to use the private DNS and NTP servers. The only difference to the `forbidden` mode is that CWNPs and Service Type LoadBalancers can be created without the restriction that only allowed networks are allowed.

Pulling container images is theoretically possible if a cluster user creates a CWNP which allows network access to an external registry. But most container registries serve the container images from large CDN networks, which have a lot of ip addresses. Simply adding the ip address of docker.io is therefore not sufficient.

Application Container Images

In order to deploy application containers into a cluster with Internet Access `forbidden` a private registry must be provided. This private registry must be located in the list of allowed networks. The DNS name of the registry must resolve in the public DNS servers. The registry must be secured with a TLS certificate that is also valid with the CA certificates from the worker node, e.g. vanilla debian ca-certificates.

Implementation

To achieve this functionality modifications have been implemented in various components in metal-stack, this includes:

Gardener Extension Provider Metal

The ControlPlane API is adopted to enable a user to configure a shoot with the internet access type `forbidden` or `restricted`. The CloudProfile can now be extended to carry the list of allowed networks, the dns and ntp servers, the registry with the mirrored registries.

ControlPlane:

```
// ControlPlaneConfig contains configuration settings for the control plane.
type ControlPlaneConfig struct {
    metav1.TypeMeta

    // NetworkAccessType defines how the cluster can reach external networks.
    // +optional
    NetworkAccessType *NetworkAccessType
}
type (
    // NetworkAccessType defines how a cluster is capable of accessing external networks
    NetworkAccessType string
)

const (
    // NetworkAccessBaseline allows the cluster to access external networks in a baseline manner
    NetworkAccessBaseline = NetworkAccessType("baseline")
    // NetworkAccessRestricted access to external networks is by default restricted to registries,
dns and ntp to partition only destinations.
```

```
    // Therefore registries, dns and ntp destinations must be specified in the cloud-profile
accordingly.
    // If this is not the case, restricting the access must not be possible.
    // Image overrides for all images which are required to create such a shoot, must be specified.
No other images are provided in the given registry.
    // customers can define own rules to access external networks as in the baseline.
    // Service type LoadBalancers are also not restricted.
    NetworkAccessRestricted = NetworkAccessType("restricted")
    // NetworkAccessForbidden in this configuration a customer can no longer create rules to access
external networks.
    // which are outside of a given list of allowed networks. This is enforced by the firewall.
    // Service type LoadBalancers are also not possible to open a service ip which is not in the
list of allowed networks.
    // This is also enforced by the firewall.
    NetworkAccessForbidden = NetworkAccessType("forbidden")
)
```

A sample Shoot Spec:

```
---
apiVersion: core.gardener.cloud/v1beta1
kind: Shoot
metadata:
  name: isolated
  namespace: sample
spec:
  provider:
    type: metal
      controlPlaneConfig:
        networkAccessType: forbidden
...
```

CloudProfile:

```
type NetworkIsolation struct {
    // AllowedNetworks is a list of networks which are allowed to connect in restricted or forbidden
NetworkIsolated clusters.
    AllowedNetworks AllowedNetworks
    // DNSServers
    DNSServers []string
    // NTPServers
    NTPServers []string
    // The registry which serves the images required to create a shoot.
    RegistryMirrors []RegistryMirror
}

// AllowedNetworks is a list of networks which are allowed to connect in restricted or forbidden
NetworkIsolated clusters.
type AllowedNetworks struct {
    // Ingress defines a list of networks which are allowed for incoming traffic like service type
LoadBalancer
    // to allow all you must specify 0.0.0.0/0 or ::/0
    Ingress []string
    // Egress defines a list of networks which are allowed for outgoing traffic
    // to allow all you must specify 0.0.0.0/0 or ::/0
    Egress []string
}
```

```
type RegistryMirror struct {
    // Name describes this server
    Name string
    // Endpoint is typically the url of the registry in the form https://hostname
    Endpoint string
    // IP is the ipv4 or ipv6 address of this server
    IP string
    // Port at which port the service is reachable
    Port int32
    // This Registry Mirror mirrors the following registries
    MirrorOf []string
}
```

A sample configuration in the CloudProfile would look like:

```
  network-isolation:
    allowedNetworks:
      egress:
      - 1.2.3.0/24 # Internet CIDR of the Provider
      - 100.64.0.0/10
      - 10.0.0.0/8
      ingress:
      - 100.64.0.0/10
    dnsServers:
      - "1.2.3.1"
      - "1.2.3.2"
      - "1.2.3.3"
    ntpServers:
      - "1.2.3.1"
      - "1.2.3.2"
      - "1.2.3.3"
    registryMirrors:
      - name: test registry
        endpoint: https://some.private.registry
        ip: "1.2.3.4"
        port: 443
        mirrorOf:
            - "docker.io"
            - "quay.io"
            - "eu.gcr.io"
            - "ghcr.io"
            - "registry.k8s.io"
```

The GEPM generates machine classes for the MCM that contain the NTP and DNS configuration for the machine. The machine-controller-manager-provider-metal implements machine creation containing these properties through the metal-api.

OS Metal Extension

Based on the configuration of a cluster the configuration of the containerd must be changed to pull images from the private registry mirror.

If a cluster is either configured with `restricted` or `forbidden` and for every registry mirror an additional `certs.d/$HOST/hosts.yaml` will be created. This is in line with Gardener's containerd Registry

Configuration.

```
# certs.d/docker.io/hosts.yaml

server = "https://docker.io"
[host."https://some.private.registry"]
  capabilities = ["pull", "resolve"]
```

Firewall Controller Manager and Firewall Controller

The Firewall Controller Manager has extended the FirewallSpec to configure the Firewall Controller which must enforce the restrictions regarding allowed networks.

```
// FirewallSpec defines parameters for the firewall creation along with configuration for the
firewall-controller.
type FirewallSpec struct {
    // AllowedNetworks defines which networks are allowed to connect to, and allow incoming traffic
from.
    // Is enforced with NetworkAccessForbidden.
    // The node network is always allowed.
    AllowedNetworks AllowedNetworks `json:"allowedNetworks,omitempty"`
}

// AllowedNetworks is a list of networks which are allowed to connect when NetworkAccessType is
NetworkAccessForbidden.
type AllowedNetworks struct {
    // Ingress defines a list of cidrs which are allowed for incoming traffic like service type
LoadBalancer
    Ingress []string `json:"ingress,omitempty"`
    // Egress defines a list of cidrs which are allowed for outgoing traffic
    Egress []string `json:"egress,omitempty"`
}
```

Also the ClusterwideNetworkPolicy in the Firewall Controller was changed to show the deployment status of a CWNP.

```


type ClusterwideNetworkPolicy struct {
    metav1.TypeMeta   `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec   PolicySpec   `json:"spec,omitempty"`
    Status PolicyStatus `json:"status,omitempty"`
}

// PolicyDeploymentState describes the state of a CWNP deployment
type PolicyDeploymentState string

const (
    // PolicyDeploymentStateDeployed the CWNP was deployed to a native nftable rule
    PolicyDeploymentStateDeployed = PolicyDeploymentState("deployed")
    // PolicyDeploymentStateIgnored the CWNP was not deployed to a native nftable rule because it is
outside of the allowed networks
    PolicyDeploymentStateIgnored  = PolicyDeploymentState("ignored")
```

```
)

// PolicyStatus defines the observed state for CWNP resource
type PolicyStatus struct {
    // FQDNState stores mapping from FQDN rules to nftables sets used for a firewall rule.
    // Key is either MatchName or MatchPattern
    // +optional
    FQDNState FQDNState `json:"fqdn_state,omitempty"`
    // State of the CWNP, can be either deployed or ignored
    State PolicyDeploymentState `json:"state"`
    // Message describe why the state changed
    Message string `json:"message,omitempty"`
}
```

Cloud Controller Manager

This component was adopted to allow to be started without a default network specified. This was actually always the internet network and if no ip address was specified in the Service Type LoadBalancer, one ip was allocated from this default network. For isolated clusters this is not provided and a cluster user must always specify this ip to get a working load balancer.

OCI Mirror

The OCI Mirror is a new application which acts as a scheduled job that pulls a given list of container images and pushes them to a private registry (which will then serve as the private registry mirror). The detailed description can be read on the project website.

Related Pull Requests

- Gardener Extension Provider
- Firewall Controller Manager
- Firewall Controller
- OS Metal Extension
- Metal Cloud Controller Manager
- Metal Networker
- Metal Images
- OCI Mirror

GPU Workers

For workloads which require the assistance of GPUs, support for GPUs in bare metal servers was added to metal-stack.io v0.18.0.

GPU Operator installation

With the nvidia image a worker has basic GPU support. This means that the required kernel driver, the containerd shim and the required containerd configuration are already installed and configured.

To enable `Pods` that require GPU support to be scheduled on a worker node with a GPU, a `gpu-operator' must be installed. This has to be done by the cluster owner after the cluster is up and running.

The simplest way to install this operator is as follows:

```
helm repo add nvidia https://helm.ngc.nvidia.com/nvidia
helm repo update

kubectl create ns gpu-operator
kubectl label --overwrite ns gpu-operator pod-security.kubernetes.io/enforce=privileged

helm install --wait \
  --generate-name \
  --namespace gpu-operator \
  --create-namespace \
    nvidia/gpu-operator \
    --set driver.enabled=false \
    --set toolkit.enabled=true
```

After that `kubectl describe node` must show the gpu in the capacity like so:

```
...
Capacity:
  cpu:                64
  ephemeral-storage:  100205640Ki
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             263802860Ki
  nvidia.com/gpu:     1
  pods:               510
...
```

With this basic installation, the worker node is ready to process GPU workloads.

> **WARNING**
>
> However, there is a caveat - only one 'Pod' can access the GPU. If this is all you need, no additional configuration is required. On the other hand, if you are planning to deploy multiple applications that

> require GPU support, and there are not that many GPUs available, you will need to configure the `gpu-operator` to allow the GPU to be shared between multiple `Pods`.

There are several approaches to sharing GPUs, please consult the official Nvidia documentation for further reference.

https://developer.nvidia.com/blog/improving-gpu-utilization-in-kubernetes
https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/gpu-operator-mig.html
https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/gpu-sharing.html

With this, happy AI processing.

Storage

When working with bare-metal servers, providing cloud storage is a challenge. With physical machines there is no opportunity that a hypervisor can mount storage devices into the servers and thus, we have to implement other mechanisms that are capable of dynamically mounting storage onto the machines.

In the meantime, we have started to integrate third-party solutions into our metal-stack landscape. They help us to provide modern, well-integrated and scalable storage solutions to our end-users.

Lightbits Labs NVMe over TCP Storage Integration

Lightbits Labs offers a proprietary implementation of persistent storage using NVMe over TCP. The solution has some very superior traits that fit very well to metal-stack. The strongest advantages are:

- High performance
- Built-in multi-tenant capabilities
- Configurable compression and replication factors

We are maintaining an open source integration for running LightOS in our Gardener cluster provisioning. You can enable it through the controller registration of the gardener-extension-provider-metal.

With the integration in place, the extension-provider deploys a duros-controller along with a Duros Storage CRD into the seed's shoot namespace. The duros-controller takes care of creating projects and managing credentials at the Lightbits Duros API. It also provides storage classes as configured in the extension-provider's controller registration to the customer's shoot cluster such that users can start consuming the Lightbits storage immediately.

Simple Node Local Storage with csi-driver-lvm

If you wish to quickly start off with cluster provisioning without caring so much about complex cloud storage solutions, we recommend using a small storage driver we wrote called csi-driver-lvm. It provides a storage class that manages node-local storage through LVM.

A definition of a PVC can look like this:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: csi-pvc
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
```

```
      storage: 100Mi
  storageClassName: csi-driver-lvm-linear
```

The solution does not provide cloud-storage or whatsoever, but it improves the user's accessibility of local storage on bare-metal machines through Kubernetes. Check out the driver's documentation here.

Security Principles

metal-stack adheres to several security principles to ensure the integrity, confidentiality and availability of its services and data. These principles guide the design and implementation of security measures across the metal-stack architecture.

Minimal Need to Know

The minimal need to know principle is a security concept that restricts access to information and resources only to those who absolutely need it for their specific role or task. This principle is implemented throughout the metal-stack architecture and operational practices to enhance security and reduce the risk of unauthorized access or data breaches.

RBAC

> **INFO**
>
> As of now metal-stack does not implement fine-grained Role-Based Access Control (RBAC) within the `metal-api` but this is worked on in MEP-4.

As described in our User Management concept the metal-api currently offers three different user roles for authorization:

- `Admin`
- `Edit`
- `View`

To ensure that internal components interact securely with the metal-api, metal-stack assigns specific roles to each service based on the principle of least privilege.

Component	Role
metal-image-cache-sync	View
machine-controller-manager-provider-metal	Edit
gardener-extension-provider-metal	Edit
metal-bmc	Edit
metal-core	Edit
metal-hammer	View

Component	Role
metal-metrics-exporter	View
metal-ccm	Admin
pixiecore	View
metal-console	Admin
cluster-api-provider-metal-stack	Edit
firewall-controller-manager	Edit

Users can interact with the metal-api using metalctl, the command-line interface provided by metal-stack. Depending on the required operations, users should authenticate with the appropriate role to match their level of access.

Defense in Depth

Defense in depth is a security strategy that employs multiple layers of defense to protect systems and data. By implementing various security measures at different levels, metal-stack aims to mitigate risks and enhance overall security posture.

Redundancy

Redundancy is a key principle in metal-stack's security architecture. It involves duplicating critical components and services to ensure that if one fails, others can take over, maintaining system availability and reliability. This is particularly important for data storage and processing, where redundancy helps prevent data loss and ensures continuous operation.

BMC User Management

For bare metal provisioning with metal-stack, two dedicated users to interact with a machine BMC are created. The `metal-hammer` first creates a BMC user called `root` or `superuser` with the administrator privilege. The password used, is configured with the Ansible variable `metal_api_bmc_superuser_pwd`. It is necessary e.g. for `metal-bmc`, to perform its actions while deleting a machine and adding it to the pool of available machines again. Since metal-stack operates within a Kubernetes cluster, the `metal_api_bmc_superuser_pwd` is stored as a Kubernetes Secret in the metal-control-plane namespace, with the value represented in base64 encoding.

> **INFO**

> Note: The superuser feature is optional. If no superuser password is configured, it is disabled. In this case, `metal-bmc` cannot report machine data for unallocated machines.

Afterwards a user called `metal` with administrator privileges is created by `metal-hammer`. When a machine registers with the `metal-api`, a password for the metal user is automatically generated and not set through an Ansible variable in `metal-roles`. This password is added to the machine's IPMI details, which are then persisted in the `metal-db`. The register event triggers the `metal-api` to store all required machine information. Each time a machine is allocated, the password is reset and updated in the database, while the user account itself remains unchanged. The applied password constraints are as follows:

- Password length: 10 characters
- Number of digits: 3
- Number of special characters: 0
- Uppercase allowed
- Repeated characters allowed

> **INFO**
>
> Note: The `metal-db` is not encrypted. Access to the metal-control-plane namespace should therefore be carefully restricted to trusted administrators only.

The `metal` user is solely intended for SOL (Serial over LAN) out-of-band administrative access to the machine via metalctl. To establish this connection, the `metal-console` component is used, which transfers console output over SSH. This setup ensures secure, remote out-of-band management, allowing operators to troubleshoot and control machines even when the operating system is unavailable. To maintain security, the BMC credentials should be treated as system-managed accounts. For security and compliance, administrators are strongly advised to avoid interactive logins with them.

SBOM

Every container image and binary that's part of metal-stack contains an *SBOM* (Software Bill of Materials). It provides a detailed inventory of components within container images and binaries, enabling you to manage vulnerabilities and compliance effectively.

We decided to use *SPDX* (Software Package Data Exchange), as it is among the most widely adopted standards and is natively supported in Docker. Docker utilizes the in-toto SPDX format, while binary-*SBOM*s are created using Syft.

*SBOM*s are created as part of each repository's *GitHub Actions* workflow utilizing Anchore SBOM Action for binaries and Build and push Docker images for container images.

Download *SBOM* of a container image

```
docker buildx imagetools inspect ghcr.io/metal-stack/<image name>:<tag> --format "{{ json .SBOM.SPDX }}" > sbom.json
```

For further info, refer to the Docker docs.

Download *SBOM* of a binary from the GitHub release

```
wget https://github.com/metal-stack/<repository name>/releases/latest/download/sbom.json
```

Please note, if more than one binary is released, e.g. for different platforms / architectures, you are required to include this info in the *SBOM* file name as well.

```
# This is an example using https://github.com/metal-stack/metalctl
wget https://github.com/metal-stack/metalctl/releases/latest/download/sbom-darwin-arm64.json
```

Identify CVEs

There are many tools that can help you to identify the CVEs with the help of an SBOM. Just to name one example, grype can be used to do this, which would look like this:

```
$ grype sbom-darwin-arm64.json
 ✔ Scanned for vulnerabilities     [14 vulnerability matches]
   ├── by severity: 0 critical, 5 high, 9 medium, 0 low, 0 negligible
NAME                              INSTALLED  FIXED IN         TYPE       VULNERABILITY
SEVERITY  EPSS          RISK
stdlib                            go1.24.5   1.24.8, 1.25.2   go-module  CVE-2025-61723
```

```
High     < 0.1% (23rd)  < 0.1
stdlib                              go1.24.5   1.24.8, 1.25.2   go-module  CVE-2025-61725
High     < 0.1% (23rd)  < 0.1
stdlib                              go1.24.5   1.24.8, 1.25.2   go-module  CVE-2025-58186
Medium   < 0.1% (17th)  < 0.1
stdlib                              go1.24.5   1.24.8, 1.25.2   go-module  CVE-2025-61724
Medium   < 0.1% (17th)  < 0.1
stdlib                              go1.24.5   1.24.8, 1.25.2   go-module  CVE-2025-47912
Medium   < 0.1% (16th)  < 0.1
stdlib                              go1.24.5   1.24.8, 1.25.2   go-module  CVE-2025-58188
High     < 0.1% (8th)   < 0.1
stdlib                              go1.24.5   1.24.8, 1.25.2   go-module  CVE-2025-58189
Medium   < 0.1% (12th)  < 0.1
github.com/gorilla/csrf             v1.7.3                      go-module  GHSA-82ff-hg59-8x73
Medium   < 0.1% (8th)   < 0.1
stdlib                              go1.24.5   1.23.12, 1.24.6  go-module  CVE-2025-47907
High     < 0.1% (4th)   < 0.1
stdlib                              go1.24.5   1.23.12, 1.24.6  go-module  CVE-2025-47906
Medium   < 0.1% (5th)   < 0.1
stdlib                              go1.24.5   1.24.8, 1.25.2   go-module  CVE-2025-58185
Medium   < 0.1% (6th)   < 0.1
stdlib                              go1.24.5   1.24.9, 1.25.3   go-module  CVE-2025-58187
High     < 0.1% (2nd)   < 0.1
stdlib                              go1.24.5   1.24.8, 1.25.2   go-module  CVE-2025-58183
Medium   < 0.1% (2nd)   < 0.1
github.com/go-viper/mapstructure/v2  v2.3.0     2.4.0            go-module  GHSA-2464-8j7c-4cjm
Medium   N/A            N/A
```

Or even simpler by passing the output of `docker buildx imagetools inspect` into grype like so:

```
docker buildx imagetools inspect ghcr.io/metal-stack/<image name>:<tag> --format "{{ json .SBOM.SPDX }}" | grype
```

Cryptography

metal-stack incorporates multiple layers of cryptographic protection and secure communication to ensure system integrity and confidentiality:

TLS Certificate Management

TLS certificates used by metal-stack components - as outlined in the architecture section - can be generated using either RSA 4096-bit or ECDSA 256-bit keys. We recommend RSA 4096.

By default, in-cluster communication is not encrypted. If encryption is required within the cluster, it must be configured manually using a service mesh (e.g., Istio or Linkerd) or a similar mechanism. For outbound traffic, we recommend integrating cert-manager in combination with Let's Encrypt to handle certificate issuance and enable automated certificate rotation for ingress domains. In offline environments where Let's Encrypt cannot be used, the certificates must be issued and managed manually or via an internal CA.

VPN & Network Encryption

metal-stack employs WireGuard-based VPN technology, orchestrated via Headscale. WireGuard leverages Elliptic Curve Cryptography (ECC) for key exchange and relies on the Noise Protocol Framework to establish secure and lightweight cryptographic handshakes.

Authentication with JWT

Access to the `metal-api` is protected using JWT (JSON Web Tokens). These tokens are generated and verified using the `go-jose` library, which implements JOSE standards.

Supported signature algorithms include:

- RSA (RS256, RS384, RS512)
- RSA-PSS (PS256, PS384, PS512)
- ECDSA (ES256, ES384, ES512)
- EdDSA

Communication Matrix

This matrix describes the communication between components in the metal-stack and their respective security properties. Please note that depending on your setup and configuration, some components may not be present, may have different security properties and might communicate differently than described here. The communication processes described here correspond to the standard configuration and setup.

Legend:

- `C`: Confidentiality, cryptography, encryption. Marked with an `x` if the communication is encrypted.
- `I`: Integrity of data. Marked with an `x` if the communication ensures data integrity.
- `Auth`: Authentication, ensures the identity of the communicating parties. Marked with an `x` if authentication is required.
- `Trust`: Only trusted networks involved. Marked with an `x` if the communication is only between trusted networks.

Plain metal-stack

While metal-stack can be used in different environments and setups, the following communication is required by metal-stack components in a standard setup. This includes all components running on the control plane, partition management and machines.

> **INFO**
>
> Description The following table might not be displayed in completeness. Scroll to the right to see all entries.

No.	Component	Source Zone	Protocol	Destination	Destination Zone	Port	C	I	Auth	Tru
1.1	metalctl	Internet	HTTPS	metal-api	Metal Control Plane	443	x	x	x	
1.2	metalctl	Internet	HTTPS	OIDC Provider	unknown	443	x	x	x	
1.3	metalctl	Internet	HTTPS	GitHub	Internet	443	x	x		
2.1	metal-api	Metal Control Plane	TCP	metal-db	Metal Control Plane	28015			x	

No.	Component	Source Zone	Protocol	Destination	Destination Zone	Port	C	I	Auth	Tru
2.2	metal-api	Metal Control Plane	TCP	masterdata-api	Metal Control Plane	8443			x	x
2.3	metal-api	Metal Control Plane	HTTP	ipam	Metal Control Plane	9090				x
2.4	metal-api	Metal Control Plane	TLS	nsq	Metal Control Plane	4150	x	x	x	x
2.5	metal-api	Metal Control Plane	HTTP	nsq lookupd	Metal Control Plane	4161			x	x
2.6	metal-api	Metal Control Plane	TCP	auditing timescaledb	Metal Control Plane	5432			x	x
2.7	metal-api	Metal Control Plane	HTTPS	headscale	Metal Control Plane	50443	x	x	x	x
2.8	metal-api	Metal Control Plane	HTTPS	S3-compatible Storage	unknown	443	?	?	?	?
2.9	metal-api	Metal Control Plane	HTTPS	OIDC Provider	unknown	443	?	?	?	?
3.1	metal-apiserver	Metal Control Plane	TCP	valkey	Metal Control Plane	6379			x	x
3.2	metal-apiserver	Metal Control Plane	TCP	metal-db	Metal Control Plane	28015	x	x	x	x
3.3	metal-apiserver	Metal Control Plane	TCP	masterdata-api	Metal Control Plane	8080			x	x

No.	Component	Source Zone	Protocol	Destination	Destination Zone	Port	C	I	Auth	Tru
3.4	metal-apiserver	Metal Control Plane	HTTP	ipam	Metal Control Plane	9090				x
3.5	metal-apiserver	Metal Control Plane	TCP	auditing timescaledb	Metal Control Plane	5432			x	x
3.6	metal-apiserver	Metal Control Plane	HTTPS	headscale	Metal Control Plane	50443	x	x	x	x
3.7	metal-apiserver	Metal Control Plane	HTTPS	OIDC Provider	unknown	443	x	x	x	1
4.1	masterdata-api	Metal Control Plane	TCP	masterdata-db	Metal Control Plane	5432			x	x
5.1	ipam	Metal Control Plane	TCP	ipam-db	Metal Control Plane	5432			x	x
6.1	backup-restore-sidecar	Metal Control Plane	HTTPS	S3-compatible Storage	unknown	443	?	?	?	1
6.2	backup-restore-sidecar	Metal Control Plane	HTTPS	Google API	Internet	443	x	x	x	
6.3	backup-restore-sidecar	Metal Control Plane	TCP	Postgres	Metal Control Plane	5432			x	x
6.4	backup-restore-sidecar	Metal Control Plane	TCP	RethinkDB	Metal Control Plane	28015			x	x
6.5	backup-restore-sidecar	Metal Control Plane	TCP	ETCD	Metal Control Plane	2380			x	x
6.6	backup-restore-sidecar	Metal Control Plane	TCP	Redis	Metal Control Plane	6379			x	x

No.	Component	Source Zone	Protocol	Destination	Destination Zone	Port	C	I	Auth	Tru
6.7	backup-restore-sidecar	Metal Control Plane	TCP	keydb	Metal Control Plane	6379			x	x
7.1	metal-console	Partition Management	HTTP	metal-api	Metal Control Plane	8080			x	x
7.2	metal-console	Partition Management	HTTPS	metal-bmc	Partition Management	3333	x	x	x	x
8.1	ssh	unknown	TCP	metal-console	Partition Management	10001	x	x	x	1
9.1	pixiecore	Partition Management	HTTPS	metal-api	Metal Control Plane	443	x	x	x	x
10.1	metal-bmc	Partition Management	HTTPS	metal-api	Metal Control Plane	443	x	x	x	x
10.2	metal-bmc	Partition Management	TLS	nsq	Partition Management	4150	x	x	x	x
10.2	metal-bmc	Partition Management	IPMI	machine BMC	Machine	623			x	x
11.1	metal-cache-image-sync	Partition Management	HTTPS	S3-compatible Storage	unknown	443	?	?	?	
11.2	metal-cache-image-sync	Partition Management	HTTPS	metal-api	Metal Control Plane	443	x	x	x	
12.1	metal-hammer	Machine	HTTPS	metal-api	Metal Control Plane	443	x	x	x	x

No.	Component	Source Zone	Protocol	Destination	Destination Zone	Port	C	I	Auth	Tru
12.2	metal-hammer	Machine	HTTPS	pixiecore	Partition Management	443	x	x		x
12.3	metal-hammer	Machine	HTTPS	Prometheus	unknown	443	x	x	x	x
12.4	metal-hammer	Machine	HTTP	HAProxy	Metal Control Plane	9001		x		x
12.5	metal-hammer	Machine	HTTPS	Container Registry	internet	443	x	x	?	
13.1	machine firmware	Machine	HTTPS	pixiecore	Partition Management	443	x	x		x
13.2	machine firmware	Machine	TFTP	pixiecore	Partition Management	69				x
14.1	machine OS	Machine	DHCP	DHCP Server	Machine	67/68				x
14.2	machine OS	Machine	DNS	DNS Server	Machine	53				x
14.3	machine OS	Machine	NTP	NTP Server	Machine	123				x
15.1	metal-metrics-exporter	Metal Control Plane	HTTPS	metal-api	Metal Control Plane	443	x	x	x	x
16.1	prometheus	Metal Control Plane	HTTPS	metal-api	Metal Control Plane	443	x	x	x	x

No.	Component	Source Zone	Protocol	Destination	Destination Zone	Port	C	I	Auth	Tru
16.2	prometheus	Metal Control Plane	HTTPS	metal-metrics-exporter	Metal Control Plane	9080				x
16.3	prometheus	Metal Control Plane	HTTPS	metal-apiserver	Metal Control Plane	443	x	x	x	x
16.4	prometheus	Metal Control Plane	HTTPS	masterdata-api	Metal Control Plane	2113	x	x	x	x

Used Technologies

Technology	Parties	Notes
DHCP	All	Used for obtaining IP addresses and boot configurations.
NTP	All	Used for synchronizing time across all components.
iPXE	Machines	Used for network-based bootstrapping of machines.
TFTP	Machines	Used for transferring boot files to machines.
HTTP	Multiple	Communication in trusted networks.
HTTPS	Multiple	Cross-network communication.
DNS	Multiple	Used for resolving hostnames to IP addresses.
Kubernetes	Cluster	Metal-stack components running in pods. Optional, but recommended.
Container Network Interface (CNI)	Kubernetes	Provides networking capabilities for pods in a cluster. Required for Kubernetes.

With SONiC

While metal-stack does not directly depend on SONiC, it is the only actively maintained implementation of our networking stack. Therefore, the following communication is required by metal-stack components to

interact with SONiC. Please note that every networking setup has its own requirements and configurations, so the following table might not be complete for your setup.

No.	Component	Source Zone	Protocol	Destination	Destination Zone	Port	C	I	Auth	Tru
S1.1	metal-core	Leaf Switches	HTTPS	metal-api	Metal Control Plane	443	x	x	x	x
S1.2	metal-core	Leaf Switches	TCP	SONiC ConfigDB Redis	Switch	6379				x
S2.1	DHCP Relay	Leaf Switches	TCP/UDP	DHCP Server	Management Server	67/68				x
S3.1	ssh client	unknown	SSH	ssh daemon	Management Server	22	x	x	x	
S3.2	ssh client	Management Server	SSH	ssh daemon	Switch	22	x	x	x	x
S4.1	FRRouting	Firewall	BGP	FRRouting	Switches	179				x
S4.2	FRRouting	Machine	BGP	FRRouting	Firewall	179				x
S4.3	FRRouting	Switches	BGP	FRRouting	Switches	179				x
S5.1	tailscale	Firewall	HTTPS	Headscale	Metal Control Plane	443	x	x	x	x

Used Technologies

Technology	Parties	Notes
VRF	Switches, Firewalls	Isolation of network segments, e.g. for management and data traffic.
VLAN	Switches, Firewalls	Layer 2 traffic segmentation.
VXLAN	Switches, Firewalls	Encapsulate Layer 2 frames in Layer 3 packets for network virtualization.
EVPN	Switches, Firewalls	Overlay network technology for scalable and flexible network architectures.
VPN	Firewalls	Management access without open SSH ports.
BGP	Multiple	Routing protocol for dynamic routing and network management.
SSH	Management Server, Switches	Secure shell access for management and configuration.
LLDP	Switches, Machines	Link Layer Discovery Protocol for network device discovery.
ICMP	Multiple	Used for network diagnostics and reachability testing.

With Gardener

When using metal-stack in conjunction with Gardener, the following communication is required by metal-stack components.

INFO

The following table might not be displayed in completeness. Scroll to the right to see all entries.

No.	Component	Source Zone	Protocol	Destination	Destination Zone	Port	C	I	Auth	Trust	Pu
G1.1	metal-ccm	Seed Cluster	HTTPS	metal-api	Metal Control Plane	443	x	x	x	x	AP Re
G1.2	metal-ccm	Seed Cluster	HTTPS	kube-apiserver	Shoot Cluster	443	x	x	x	x	AP Re
G2.1	firewall-controller-manager	Seed Cluster	HTTPS	metal-api	Metal Control Plane	443	x	x	x	x	AP Re

No.	Component	Source Zone	Protocol	Destination	Destination Zone	Port	C	I	Auth	Trust	Pu
G2.2	firewall-controller-manager	Seed Cluster	HTTPS	kube-apiserver	Seed Cluster	443	x	x	x	x	AP Re
G2.3	firewall-controller-manager	Seed Cluster	HTTPS	kube-apiserver	Shoot Cluster	443	x	x	x	x	AP Re
G3.1	firewall-controller	Firewall	HTTPS	kube-apiserver	Seed Cluster	443	x	x	x	x	AP Re
G3.2	firewall-controller	Firewall	HTTPS	kube-apiserver	Shoot Cluster	443	x	x	x	x	AP Re
G3.3	firewall-controller	Firewall	HTTPS	Controller URL	Internet	443	x	x			Se Up
G4.1	machine-controller-manager-provider-metal	Seed Cluster	HTTPS	metal-api	Metal Control Plane	443	x	x	x		AP Re
G5.1	gardener-extension-provider-metal	Seed Cluster	HTTPS	metal-api	Metal Control Plane	443	x	x	x		AP Re
G5.2	gardener-extension-provider-metal	Seed Cluster	HTTPS	kube-apiserver	Garden Cluster	443	x	x	x		AP Re
G5.3	gardener-extension-provider-metal	Seed Cluster	HTTPS	kube-apiserver	Seed Cluster	443	x	x	x		AP Re
G5.4	gardener-extension-provider-metal	Seed Cluster	HTTPS	kube-apiserver	Shoot Cluster	443	x	x	x		AP Re

Used Technologies

Technology	Parties	Notes
Gardener	Contains of multiple components.	Cluster management system for many Kubernetes.

With Cluster API

By using the Cluster API provider for metal-stack, the following communictations are required by metal-stack components.

> **INFO**
> The following table might not be displayed in completeness. Scroll to the right to see all entries.

No.	Component	Source Zone	Protocol	Destination	Destination Zone	Port	C	I	Auth	Trust
C1.1	metal-ccm	Workload Cluster	HTTPS	metal-api	Metal Control Plane	443	x	x	x	
C1.2	metal-ccm	Workload Cluster	HTTPS	kube-apiserver	Workload Cluster	443	x	x	x	x
C2.1	cluster-api-provider-metal-stack	Management Cluster	HTTPS	metal-api	Metal Control Plane	443	x	x	x	

Used Technologies

Technology	Parties	Notes
Cluster API	Contains of multiple components and additional providers.	Cluster management system for single Kubernetes clusters.

With Lightbits

In order to use Lightbits as a storage solution, the following communications are required by metal-stack components.

> **INFO**
> The following table might not be displayed in completeness. Scroll to the right to see all entries.

No.	Component	Source Zone	Protocol	Destination	Destination Zone	Port	C	I	Auth	Trust	P
L1.1	duros-controller	Seed Cluster	HTTPS	duros-api	Lightbits Cluster	443	x	x	x	x	St
L1.2	duros-controller	Seed Cluster	HTTPS	kube-apiserver	Shoot Cluster	443	x	x	x	x	Ku
L2.1	lb-csi-controller	Shoot Cluster	HTTPS	duros-api	Lightbits Cluster	443	x	x	x		St
L2.2	lb-csi-controller	Shoot Cluster	HTTPS	kube-apiserver	Shoot Cluster	443	x	x	x	x	Ku
L3.1	lb-csi-node	Shoot Cluster	TCP	duros-api	Lightbits Cluster	4420	x	x	x		St
L3.2	lb-csi-node	Shoot Cluster	TCP	duros-api	Lightbits Cluster	8009	x	x	x		St

Used Technologies

Technology	Parties	Notes
Lightbits	Storage	Used for storage solutions.

DRAFT PAGE

This page is a draft. It will only be visible in dev and be excluded from the production build.

Artifact Signing

DRAFT PAGE

This page is a draft. It will only be visible in dev and be excluded from the production build.

Integration Checks

DRAFT PAGE

This page is a draft. It will only be visible in dev and be excluded from the production build.

Network

DRAFT PAGE

This page is a draft. It will only be visible in dev and be excluded from the production build.

RBAC

The metal-api offers three different user roles for authorization:

- `Admin`
- `Edit`
- `View`

To ensure that internal components interact securely with the metal-api, metal-stack assigns specific roles to each service based on the principle of least privilege.

Component	Role
metal-image-cache-sync	View
machine-controller-manager-provider-metal	Edit
gardener-extension-provider-metal	Edit
metal-bmc	Edit
metal-core	Edit
metal-hammer	View
metal-metrics-exporter	View
metal-ccm	Admin
pixiecore	View
metal-console	Admin
cluster-api-provider-metal-stack	Edit
firewall-controller-manager	Edit

Users can interact with the metal-api using metalctl, the command-line interface provided by metal-stack. Depending on the required operations, users should authenticate with the appropriate role to match their level of access.

As part of MEP-4, significant work is underway to introduce more fine-grained access control mechanisms within metal-stack, enhancing the precision and flexibility of permission management.

Remote Access

Machines and Firewalls

Remote access to machines and firewalls is essential for performing administrative tasks such as incident management, troubleshooting and sometimes for development. Standard SSH access is often insufficient for these purposes. In many cases, direct serial console access is required to fully manage the system. metal-stack follows a security-first approach by not offering direct SSH access to machines. This practice reduces the attack surface and prevents unauthorized access that could lead to system damage. Detailed information can be found in MEP-9. Administrators can access machines in two primary ways.

Out-of-band management via SOL

```
metalctl machine console <ID> --ipmi
```

This method leverages the machine's BMC. For detailed user configuration, see the BMC User Management section.

Via metal-console:

```
metalctl machine console <ID>
```

This approach uses the `metal-console`, which is required to establish console access. This component acts as a bridge between SSH and the console protocol of the concrete machine.

Both methods ensure secure and controlled access to machines without exposing them unnecessarily to the network, maintaining the integrity and safety of the infrastructure.

Connecting directly to a machine without a clear plan of action can have unintended consequences and negatively impact stability. For this reason, administrative privileges are required. This restriction ensures that only authorized personnel with the necessary expertise can perform actions that affect the underlying infrastructure. These principles will evolve with the introduction of MEP-4.

DRAFT PAGE

This page is a draft. It will only be visible in dev and be excluded from the production build.

Security Vulnerability