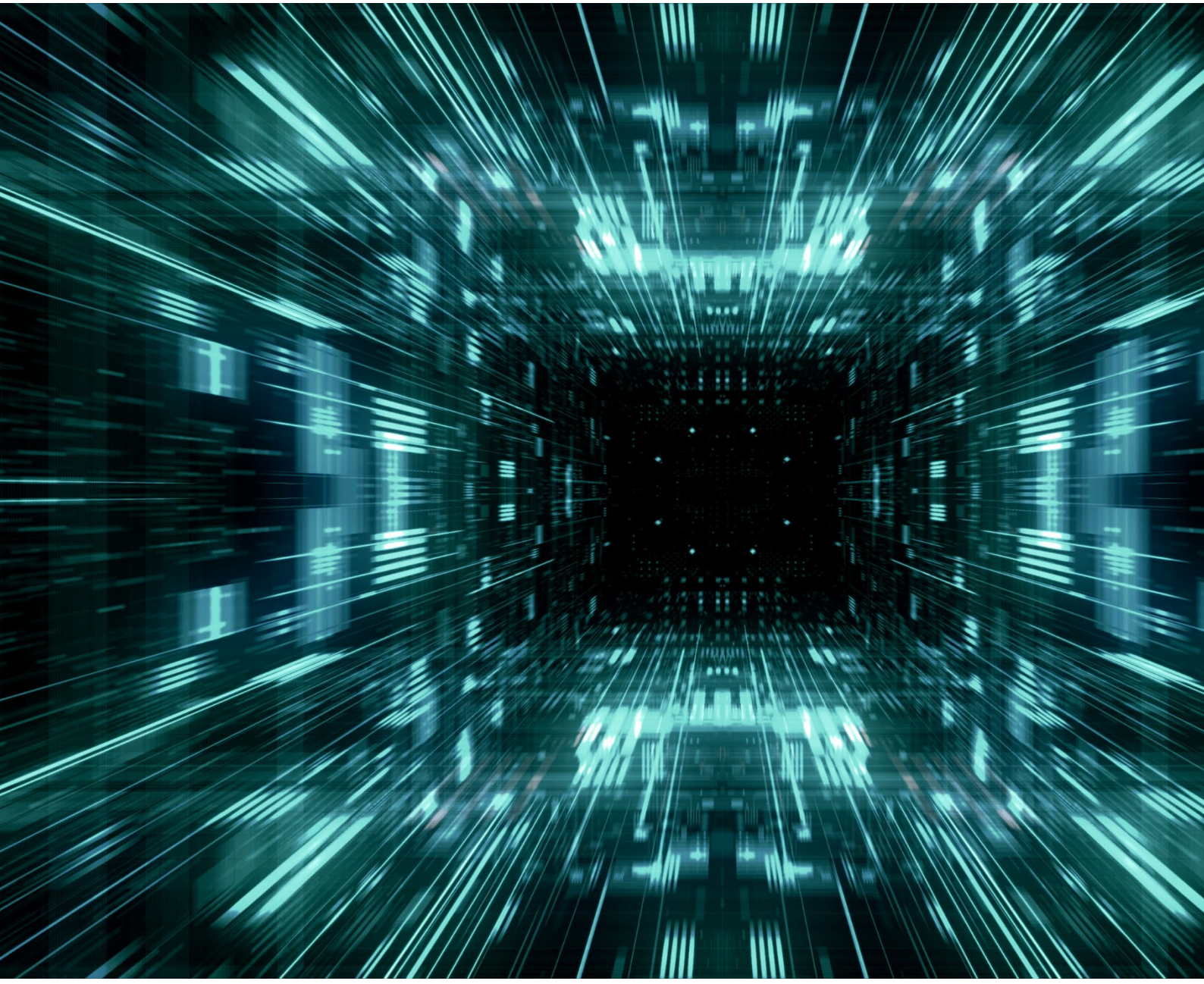
DAIM
injecting insights

# DAIM Edge Computing Platform
# **OPERATIONS MANUAL**



Revision 14, 30.05.2023



**DAIM GmbH** | Linzer Straße 2-6 | A-4320 Perg | +43 7262 52834 | office@daim.tech | ATU75416337 | FN 531357h

# 1. About This Document

This comprehensive operations manual is specifically designed for operators of the DAIM Edge Computing Platform. Its primary purpose is to provide detailed instructions on deploying, maintaining, and configuring the DAIM Management Platform, which operates in the cloud and oversees a network of Edge- and IoT devices.

# 2. Architecture

## 2.1. Overview

The **DAIM Edge Computing Platform** consists of two major building blocks:

1. The cloud-based **DAIM Management Platform** for administration of IoT devices and
2. The **DAIM Edge-OS**, a Linux Debian based software operating on the Edge-Devices.

## 2.2. DAIM Management Platform (EDP)

A Kubernetes cluster (K8s) is used to orchestrate the microservices based management platform. The platform a REST API for automating business processes and a Web-UI for manual administration and monitoring.

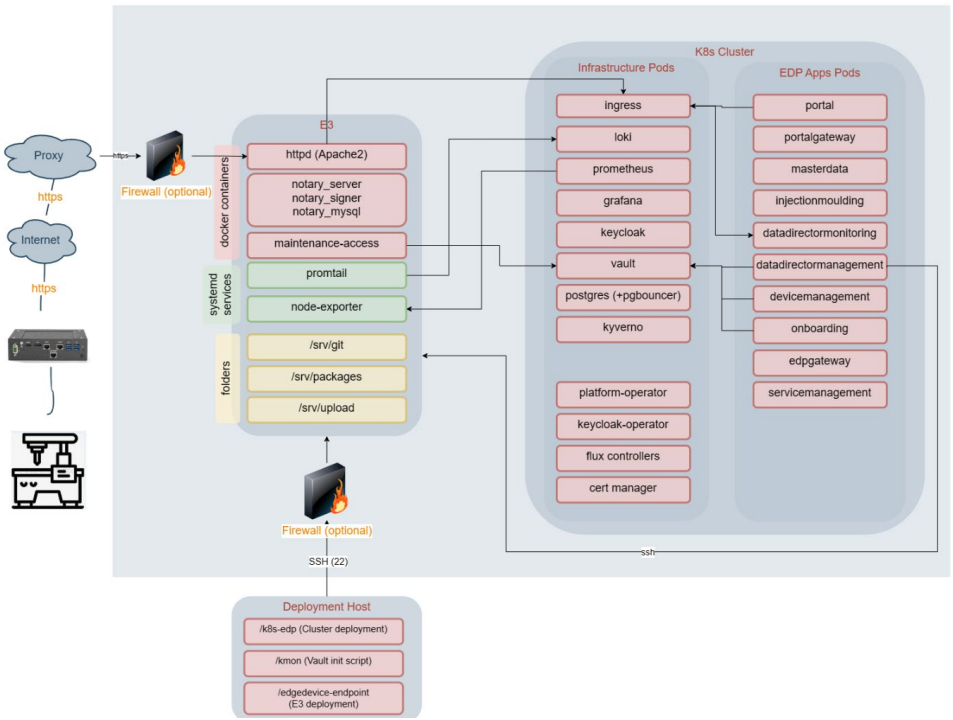



*Fig. 1 - Overview of the DAIM Edge Computing Platform*

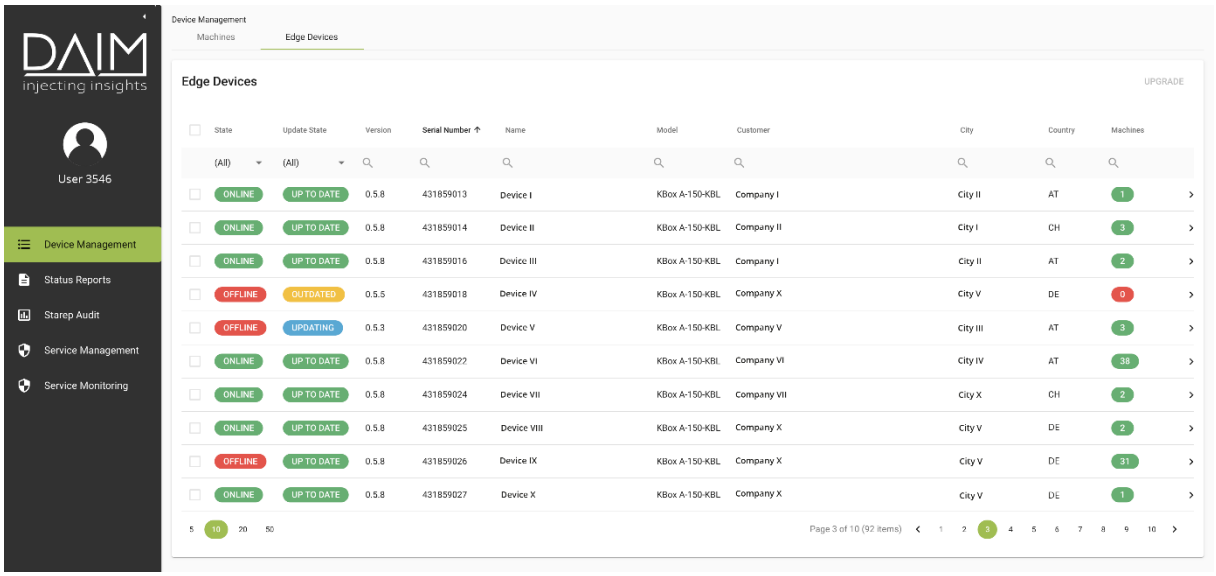



*Fig. 2 - WebUI of the DAIM Management Platform. The screenshot shows the Edge-Device management tab.*

## 2.3. DAIM Staging Concept

Usually, the platform is deployed on multiple different environments to stage future updates and releases, rather than deploying new features directly to the production environment. DAIM's recommendation is to have three distinct stages: TESTING, STAGING, and PRODUCTION:

- PRODUCTION: https://edp-portal.production.api.*<customer>*.net
- STAGING: https://edp-portal.staging.api.*<customer>*.net
- TESTING: https://edp-portal.testing.api.*<customer>*.net

Every update should first be deployed to the Testing environment. After successful tests by all stakeholders, the update is deployed to the Staging environment, where a release candidate can be tested under production similar conditions.

Once the release candidate is successfully tested, it finally can be deployed to the productive environment.

## 2.4. K8s Cluster

### 2.4.1. Infrastructure Pods

The following open-source licensed **infrastructure related** pods are available in the cluster:

- **Loki**: service log aggregation
- **Prometheus**: service metrics aggregation
- **Grafana**: dashboards for cluster monitoring
- **Kyverno**: declarative policy management

- **Vault**: Hashicorp vault for Edge-Device certificate management
- **Keycloak**: Identity and Access Management service used for authentication and authorization
- **Postgres**: relational database server
    - Backups are created by a CronJob named “pgbackup” that is running scheduled “0 */3 * * *” (every 3 hours). This job dumps all postgres databases and stores the dumps to the OSS bucket “edp-postgres-backup” on endpoint “oss-eu-central-1-internal.aliyuncs.com” in a separate folder for every environment (testing, staging, production)
    - The OSS bucket “edp-postgres-backup” must be configured to remove outdated dumps (e.g. older that 7 days).
- **Nginx Ingress Controller**: Load balancer for accessing services running in the cluster
- **Kubernetes Operators**:
    - Platform-operator: deploy DAIM micro-services and their dependencies
    - Keycloak-operator: deploy keycloak entities like clients, users or realms
    - Flux-controllers: deploy cluster components by reading manifests from the OSS bucket “edp-manifests”
    - Cert-manager: create/update SSL certificates

### 2.4.2. EDP Pods

This services are part of the DAIM technology stack and are part of the Management Platform

- **Portalgateway**: API gateway for external requests
- **Masterdata**: service to manage customers and users
- **Injectionmoulding**: service for managing machines
- **Datadirectormonitoring**: service for heartbeat, notification, websocket management
- **Datadirectormanagement**: service for Edge-Devices, deployments, containers
- **Devicemanagement**: service providing REST API for external requests
- **Onboarding**: service for onboarding new Edge-Devices (create client certificates)
- **Edpgateway**: service providing REST API for Management Web UI
- **Servicemanagement**: internal service registry (Eureka, Spring Boot Admin)
- **Portal**: Device Management Web UI (Angular Application with NGINX)

## 2.5. E3 Endpoint Host

In addition to the K8s cluster, there is also a dedicated host outside the cluster for communication with the Edge-Devices via the internet. This host is called “E3” (=**E**xternal **E**dge-device **E**ndpoint). This host is the single endpoint for all requests that Edge-Devices send to the platform. All Edge-Devices are allowed to communicate with the E3 server only.

All requests are handled by an Apache2 docker container on port 443 (HTTPS). Each request is validated using SSL client certificate validation except for requests to the “/onboarding/**” URL. These requests manage the onboarding of new Edge-Devices in order to download client certificates from the backend to the Edge-Device.

In case an **Edge-Device is removed** from the platform it’s certificate will be revoked and added to the Certificate Revocation List (CRL) that Apache2 is using for client certificate validation.

## 2.6. Deployment Host

For maintenance issues there is another dedicated host that provides all necessary tools, credentials and checked out repositories for accessing the EDP infrastructure.

## 2.7. Edge-Devices

An Edge-Device is the physical entity used to connect any IoT device (usually a production machine or a plant) to any backend service or platform. These devices are provisioned using the **DAIM Install Server** either during the production process by the device manufacturer or by the device operator.

The install server is also available as VDI (VirtualBox image) for setting up Edge-Devices in a specific development mode.

After provisioning, the Edge-Devices do not contain any credentials for accessing the Management Platform EDP. They must be initially onboarded before they can be used. For a successful onboarding an internet connection is required

The onboarding is done via the Edge-Device configuration web frontend by connecting a computer to the support LAN port of the device. On this network interface the devices host a web server with the configuration UI.

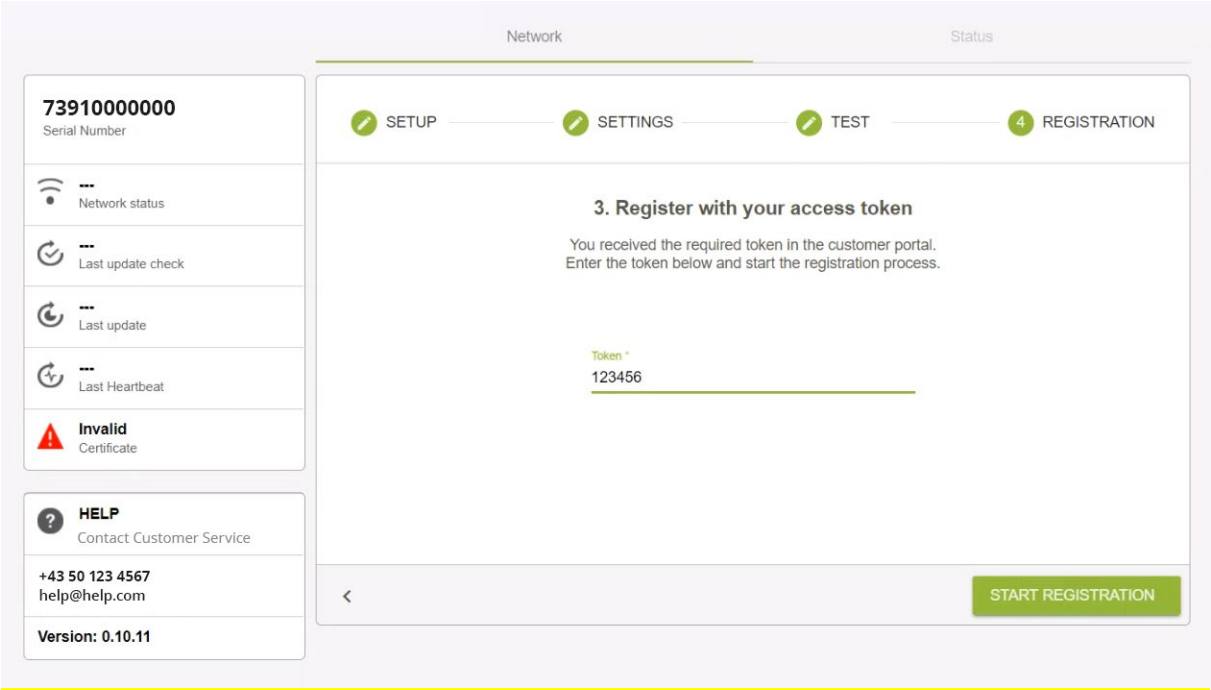



*Fig. 3 – Network configuration and onboarding can be done via connecting to the WebUI via LAN Support Port.*

The necessary steps to perform onboarding are described in the *Quick Start Guide* that is shipped together with the Edge-Devices. After setting up the network connection a valid token must be entered which is provided by the EDP and can be found in the Management UI.

Upon successful validation of the token, a Certificate Signing Request (CSR) is sent to the onboarding endpoint and after checking the request for validity, a client certificate is sent back to the device. This certificate is from now on used for communicating with the EDP and permits access to all E3/EDP services.

# 3. EDP Cluster Deployment

The k8s-edp git repository (https://bitbucket.org/<customer>/k8s-edp/src/master/) contains the deployment description of the DAIM Edge-Device Platform, targeting a k8s-cluster in the dedicated cloud. Flux (see https://fluxcd.io) is used for applying the manifests of the EDP cluster components, thus Flux must be installed in a preliminary step on the cluster.

The latest information for the cluster deployment is checked in to the repository as README.md.

## 3.1. Setup deployment environment

This section describes the preliminary setup that is required in order to perform cluster deployments on Kubernetes.

### 3.1.1. Tools

In order to execute the init-scripts and publish the manifests these tools are needed:

- kubectl (https://kubernetes.io/docs/tasks/tools/)
- flux CLI (https://fluxcd.io/docs/installation/)
- ossutil (https://partners-intl.aliyun.com/help/doc-detail/120075.htm)
- gnupg
- sops (https://github.com/mozilla/sops)
- ansible 2.10+ (pip install ansible)
- helm (https://helm.sh)
- watch

Configure kubectl and ossutil according to their documentation with a user that has the necessary privileges on the cluster and OSS Bucket.

Ansible must be installed in version 2.10+. In Ubuntu, it is necessary to remove the apt package and install it via pip:

```
pip install ansible
pip install kubernetes
# open a new shell to use updated symlinks
bash
```

## 3.2. Secret management

Flux uses a GPG key pair for encrypting/decrypting secrets in Kubernetes. The secret contents in the Bucket need to be encrypted with the GPG public key using a tool called sops. When the secrets are

fetched by flux, flux needs the corresponding GPG private key for decryption. Thus, a secret containing the private key needs to be added to the cluster before the deployment can start. For each environment, the public key and a matching sops-configuration needs to be present in the environment-specific secret-folder (secrets/<environment>).

### 3.2.1. Create the GPG key-pair

The key-pair for encrypting the cluster secrets needs to be created before initialization. The private key needs to be stored in the cluster-secret named "cluster-sops-gpg" in the namespace "flux-system". The public key goes to the folder containing the environment-specific secrets (secrets/<environment>).

The script **create-gpg.sh** creates a key-pair and also creates the necessary files which need to be copied into the secret-folder. The script needs a key-name and comment which need to be provided as arguments. If a key with the given name already exists it will not be overwritten.

Additionally, an output-path needs to be provided where the script will create all necessary files. But be aware that the created secret contains the GPG private key and therefore must NOT be committed into any VCS.

```
create-gpg.sh "EDP-staging" "encrypting secrets for flux" .secrets/gpg
```

The output directory contains three files:

- .sops.pub.asc - GPG public key
- .sops.yaml - configuration for sops, so it knows which key to use for encryption
- secret.yaml - secret containing the GPG private key

The ".sops*"-files need to be copied into the environment-specific secret-folder "secrets/<environment>". The **secret.yaml** is applied to the cluster by the **init_cluster.sh** script.

### 3.2.2. Create secrets manifests

The script create-secrets.sh creates and encrypts all needed secrets to the environment specific secrets-folder. These only contain the encrypted values of the secrets, not the unencrypted secret manifest itself, which is created using the kustomize secretGenerator.

All secrets need to be Base64-encoded strings and defined as environment variables prior to executing the create-secrets.sh (or pipeline-deploy.sh) script.

The secrets containing passwords must not contain some special characters ('\n', '\r', '@', ':', '&', '?', etc.) as this might lead to problems in the apps using the values. The create-secrets script checks for those values and will not accept them. You should use the script gen-pass.sh for creating your passwords:

```
./gen-pass.sh <length> ['base64']
```

Example to generate a 16-character password with base64 encoding:

```
./gen-pass.sh 16 base64
```

Notes:

Some passwords are used in JDBC connection strings or URLs thus all contained characters must be valid in JDBC URLs

A secret can contain multiple **environment variables** (e.g datadirectormanagement-files) that are then all available inside the pod.

| **Environment variable** | **Secret name(s)** |
|---|---|
| GRAFANA_ADMIN_PASSWORD | grafana-credentials |
| KEYCLOAK_ADMIN_PASSWORD | credential-keycloak |
| KEYCLOAK_POSTGRES_PASSWORD | keycloak-db-secret |
| POSTGRES_ADMIN_PASSWORD | postgres-admin-credentials, pgbouncer-admin-credentials |
| VAULT_DB_PASSWORD | vault-db-credentials |
| ISSUER_CA_CRT | issuer-ca |
| ISSUER_CA_TLS_KEY | issuer-ca |
| DDM_SSH_PRIVATE | datadirectormanagement-files |
| DDM_SSH_PUB | datadirectormanagement-files |
| LOKI_INGRESS_BASIC_AUTH | loki-basic-auth |

The following values are part of the GPG key the EDP uses to sign commits to Edge-Device repositories. These values are not environment-specific.

| **Environment variable** | **Secret name** |
|---|---|
| DOCKER_PULL_JSON | docker-pull |
| DDM_REV | datadirectormanagement-files |
| DDM_KEY1 | datadirectormanagement-files |
| DDM_KEY2 | datadirectormanagement-files |
| DDM_GPG_CONF | datadirectormanagement-files |
| DDM_PASSPHRASE | datadirectormanagement-files |
| DDM_PUBRING | datadirectormanagement-files |
| DDM_TRUSTDB | datadirectormanagement-files |

## 3.3. External components

All components used in the deployment can be installed from the git repositories. In order to be independent of a stable connection to those repositories, the deployment assumes that the necessary files from the repositories are available in an OSS Bucket.

The script **update-dependencies.sh** executes all the following commands and is used by the pipeline to deploy the dependencies automatically if they were changed.

These dependent repositories only need to updated when a version update of one of these components is done.

### 3.3.1. Assign Load Balancer Id and E3 IP Address

Provide the correct loadbalancer id for your environment in file *cluster-infrastructure/<environment>/nginx.patch.yaml*:

```
...
annotations:
    service.beta.kubernetes.io/alibaba-cloud-loadbalancer-id: "<loadbalancer-id>"
...
```

Provide IP address of E3 server host in file apps/<environment>/patches/global.env.patch.yaml:

```
...
- op: add
  path: /spec/podTemplate/spec/containers/0/env/0
  value:
    name: E3_HOST
    value: <e3-ip-address>
```

## 3.4. Initialize the cluster

To initialize flux for managing the cluster, the runtime components of flux and manifests describing the initial entrypoint need to be deployed manually. The script init-cluster.sh executes the necessary commands which are described in the following section.

Run shell script:

```
./init-cluster.sh <kubectl-context> <environment> <sops-gpg-secret-file>
```

The script takes three arguments:

- *kubectl-context*: Name of the context from the local kubectl-installation that should be used.
- *environment*: Which environment of the deployment should be used to manage the cluster.
- *sops-gpg-secret-file*: A file containing the secret with the private key that should be used to decrypt the secrets of the deployment. The matching public key needs to be present in the environment-specific secrets-folder, see repository folder *Secret management* for details.

### 3.4.1. Initialization steps

1. activate the kubectl-context
2. install flux using the CLI flux install
3. apply the secret containing the GPG private key
4. create the flux bucket for pulling the manifests
5. create the kustomization referencing the entry path of the deployment relative to the root of the repo (deploy\<environment>)

> ⚠ Warning: do not run the **init-cluster.sh** script on already installed clusters, as this may affect running services and components

## 3.5. Manual steps

On first-time deployment of a new cluster, there are some manual steps necessary in order to complete the initialization of several components. These steps are described here.

### 3.5.1. Initialize and Unseal Hashicorp Vault

On first cluster installation, the Hashicorp vault pod is created by the vault helm chart (see chapter 2.4.1) in namespace **vault** with name **vault-0** and started automatically. However, the vault comes uninitialized and sealed. Therefore, we provide an ansible-playbook to initialize and unseal the vault in subfolder **vault**.

Before the playbook can be executed successfully the vault-pod needs to be ready. Check the status of the vault pod with the following command:

```
kubectl get pods -n=vault vault-0 -w
```

This might take some time, as the CronJob which creates the database runs every 5 minutes, check the status of the job with the following command:

```
watch -n 1 kubectl get cronjobs.batch -n=postgres postgres-dba
```

If the column "Last Schedule" has a value the job ran at least once.

If everything is fine, go to the directory **./vault** and execute the commands described below. In order to be able to decrypt secret information we use **ansible-vault** for this manual step. Thus, there must exist a file **vault-pwd.txt** in folder **./vault** that contains the ansible vault password for the checked in ansible secret files. The content of this file is the ansible-vault password for decrypting the credential information inside the playbook. It can be obtained from DAIM and must be kept secure. (Note: for

convenience, the password file is already present on the deployment host in folder /home/devops/k8s-edp/vault).

Supported environments for the **run.sh** script are:

- testing
- staging
- production

Run the following commands to initialize the vault in your environment:

```
cd vault
ansible-galaxy collection install -f -r requirements.yml
./run.sh <environment> vault
```

This initializes the vault, fetches the root token and writes it to the vault secret that is attached to pod vault-0 (usually named vault-token-XXXXX). In a second step, this root token is used to unseal the vault for the first time. This operation reveals 3 unseal keys that are also stored in the vault secret and are used to automatically unseal the vault when the pod restarts at a later point in time. The root token and the unseal keys are mounted to the /var/run/secrets path inside the container. Automatic unsealing is done by the postStart script of the vault pod using the vault CLI that is defined as follows:

```
if [[ -f /var/run/secrets/kubernetes.io/serviceaccount/unseal_keys ]]; then
  echo $(date) waiting for vault;
  sleep 5;
  echo $(date) auto-unsealing vault;
  for line in $(cat /var/run/secrets/kubernetes.io/serviceaccount/unseal_keys); do
       vault operator unseal $line;
  done;
  vault status;
  echo $(date) vault unsealed successfully;
fi
```

Thus, after restart the vault log should contain a line showing that the unsealing was successful:

```
Wed Nov 10 13:07:57 UTC 2021 vault unsealed successfully
```

### 3.5.2. Check if deployment was successful

If all apps show status 'Ready', the deployment was successful, and the apps are running:

```
kubectl get apps -A
```

Check if vault is ready:

```
curl https://edp-service.api.testing.<customer>.internal/vault/v1/sys/health -i
```

Check the web frontends if ingress and routing is setup correctly:

- EDP Portal: https://edp-portal.api.testing.<customer>.net/device-management
  Note: On first time installation a user “admin” with password “spartan-fussy-thereby-taxpayer” is created
- Grafana: https://grafana.api.testing.<customer>.net

### 3.5.3. Restore Database Backups in Alibaba Cloud

Service databases are backed up in a CronJob “pgbackup” every 3 hours as described in chapter 1. These backups are in fact postgres database dump files that are copied to the OSS bucket “edp-postgres-backup”. Postgres dumps are created in “custom” format (=binary) so restoring can be done easily without worrying about constraints and table order.

For convenience we provide a script **restore-pgbackup.sh** in the root folder of the k8s-edp bitbucket repository (the given example restores database **masterdata** in cluster **testing**):

```
./restore-pgbackup.sh testing masterdata
```

This script performs the following commands in order to restore these dumps to the databases that are automatically created by the cluster deployment via the platform-operator from a Linux host that has access to the cluster:

```
LATEST=$(ossutil64 ls oss://edp-postgres-backup/testing/masterdata/ -s | tail -n 4 | head -n 1)
ossutil64 cp $LATEST /tmp/dump.tar --force
kubectl cp /tmp/dump.tar postgres-server-0:/tmp/dump.tar -n postgres -c postgres
kubectl exec -i -t -n postgres postgres-server-0 -c postgres -- bash -c "psql -U postgres -w -c \"ALTER
DATABASE masterdata CONNECTION LIMIT 0; SELECT pg_terminate_backend(pid) FROM pg_stat_activity WHERE
datname = 'masterdata';\""
kubectl exec -i -t -n postgres postgres-server-0 -c postgres -- bash -c "pg_restore -Fc -c -C -v -d
postgres -U postgres /tmp/dump.tar"
```

Note: You need to have a valid kubectl context pointing to the desired cluster (i.e. <customer>-testing) and you need to define valid OSS credentials in ENV for accessing the bucket to download the DB dump files (see chapter 2.2.1):

- OSS_ENDPOINT
- OSS_KEY_ID
- OSS_KEY_SECRET

### 3.5.4. EDP-K8S Folder Structure

The edp-k8s repository contains all manifests for deploying the whole K8s cluster for the Edge-Device Management Platform (EDP).

## 3.6. Update EDP Software Stack

The EDP software stack is depicted in chapter 1 on the right side as “EDP Adds Pods”. These services are managing the Edge-Devices and their deployments. The corresponding Kubernetes Deployment manifests can be found in the edp-k8s bitbucket repository in folder **apps/base**. This folder contains the App custom resource definitions for each pod. As it should be possible to have different software

versions running in different environments there is also a kustomization folder for each environment in folder **apps**:

- testing
- staging
- production

This folder contains environment-specific kustomizations for each of the EDP apps (e.g. hostnames, domain names, routes, environment variables, …). The image tags for the deployed container images are stored in the file **kustomization.yaml** in each environment folder under variable **images** with an entry for each app containing name, image and image tag. In order to upgrade the EDP software stack it is thus sufficient to update the value "newTag" for each app.

Docker images of new releases of apps are automatically pushed to the DAIM and Customer Cloud registry by the DAIM build pipeline in gitlab.unisoftwareplus.com.

New versions of apps will first be deployed to the testing environment by updating the entry in file "apps/testing/kustomization.yml" in branch testing. Pushing the changes will trigger the bitbucket pipeline and copy the changes to the OSS bucket in folder "testing". This triggers a Flux reconciliation inside the cluster which updates the modified container images and restart the corresponding pods.

During development, it is possible to use development images from DAIM artifactory in the testing environment (e.g. newName: artifactory.unisoftwareplus.com/docker-dev/datadirectormanagement, newTag: v2.4.0-2-ge4a6741), because we added the docker-pull-secret for accessing the DAIM artifactory on the testing cluster only. On clusters "staging" and "production", however, deployed images must be available in the AliCloud Container Registry <customer>-net-ecr-registry-vpc.eu-central-1.cr.aliyuncs.com/<customer>-edp-deployment.

Once tests of a new software version of one or multiple services on "testing" are successful, DAIM builds a new release version of those services and pushes the image to the customer's cloud. The new release versions are then written to images -> newTag to **apps/staging/kustomization.yaml** and the whole edp-k8s repository is merged from branch testing to branch staging (therefore, updates to the base app definitions will also be migrated to staging). This triggers the update via the bitbucket pipeline and Flux on staging to test the new release candidates in staging environment. When those tests are successful, the whole **images** list can be copied from **apps/staging/kustomization.yaml** to **apps/production/kustomization.yaml** and branch staging is merged to master. This triggers the deployment of the latest releases to the production environment.

Merging new releases from one stage to the next stage should be accomplished by a pull request in bitbucket with a manual approval by a second reviewer (4 eyes principle). The following merges should be performed:

1. Pull request merge testing → staging
2. Pull request merge staging → master

## 3.7. Uninstall

The short path to uninstalling everything is 'flux uninstall' but this does not completely clean up everything in the correct order, as some CRDs have finalizers which cannot be executed after the controller has been deleted.

> ⚠ Warning: Uninstalling a cluster cannot be undone. All existing data inside the cluster (databases, persistent volumes) will be deleted.

To correctly remove all resources from the cluster, the following steps need to be executed:

Wait a minute or so between each step, to be sure that all resource associated with the kustomization have been removed.

1. Suspend reconciliation for the main kustomization to stop flux from recreating the deleted kustomizations

```
flux suspend kustomization deploy
```

2. Remove the apps kustomization

```
flux delete kustomization apps -s
```

3. Remove the app-infrastructure kustomization

```
flux delete kustomization app-infrastructure -s
```

4. Remove the cluster-infrastructure and policies kustomization, but first check if the relevant custom resources (Apps, keycloak) have already been deleted otherwise k8s cannot call the finalizer as the controller-pods are already terminated.

```
kubectl get apps,keycloak -A
flux delete kustomization cluster-infrastructure -s
```

5. Remove the promtail-DaemonSet manually, because the pods try to connect to loki server upon termination, which might not be reachable anymore. Then remove cluster-infrastructure.

```
kubectl delete daemonset -n=loki promtail
flux delete kustomization cluster -s
```

6. Uninstall flux

```
flux uninstall -s
```

7. Wait until all namespaces except system namespaces are gone (status terminating is not gone)

```
watch -n 1 kubectl get namespaces
```

## 3.8. Flux/Kubernetes Cheat sheet

- Immediately refresh the bucket

```
flux reconcile source bucket --namespace flux-system cluster-bucket
```

- Immediately force reconciliation of kustomization

```
flux reconcile kustomization --namespace flux-system app-infrastructure
```

- Check status of the flux resources continually

```
watch -n 1 kubectl get gitrepositories,buckets,kustomizations,helmreleases -A
```

- Check the status of all apps

```
kubectl get apps -A
```

- Check if vault is unsealed

```
kubectl exec -n=vault vault-0 -- vault status
```

## 3.9. Troubleshooting

### 3.9.1. Check status of flux resources

All flux resources have a ready field, as soon as this field shows "True", everything should work as expected from a deployment point of view. This means that the necessary resources were deployed and the main pods are up and running.

If a resource displays "False" in the ready field, the status field gives a more detailed description, hinting at the source of the error:

```
kubectl get buckets,helmcharts,kustomizations,helmreleases -A
```

### 3.9.2. Known errors

- Bucket Resource: “context deadline exceeded”: The download of the bucket ran into a timeout. This should not happen regularly. If it does, the value in the field ‘timeout’ of the Bucket resource should be increased.
- Kyverno Errors: failed calling webhook \"mutate.kyverno.svc-fail\": when updating (=mutation) resources a request to kyverno is sent by a webhook definition. If however the kyverno pod is down, this fails and thus no more resource updates are possible. The only solution is to remove all kyverno webhooks at this point by running the following commands:

```
kubectl delete mutatingwebhookconfigurations/kyverno-resource-mutating-webhook-cfg
kubectl delete validatingwebhookconfigurations/kyverno-resource-validating-webhook-cfg
```

### 3.9.3. Check status of apps

The status field of an app shows if there have been any errors while reconciling the application resources or if the expected pods are not running as expected.

The following table shows the possible values of the field and its meaning

| Status | Description |
|---|---|
| reconciling | The controller is currently processing this resource and creating the k8s-resources based on the specification. |
| not ready | The controller successfully finished the reconciliation but the requested pod(s) are not ready yet. |
| ready | The requested pod(s) for this app are ready. |
| reconciliation failed | An error happened while reconciling the app manifest. In this case the controller adds events to the app resource containing more details. If this is not enough, the log of the platform-operator pod needs to be checked. |
| deleting | The App-resource was deleted and the controller is currently performing the necessary cleanup operations. |

```
kubectl get apps -A
```

### 3.9.4. Check vault status

In order for vault to provide service, the pod needs to be ready, and the vault also needs to be unsealed.The easiest way to check the status is calling the vault cli, which is installed in the vault container.

```
kubectl exec -n=vault vault-0 -- vault status
```

The output shows the status of the vault server including the value "Sealed". If this field is true, the automatic unsealing upon startup of the pod did not work. To analyse the reasons, the log of the container needs to be checked.

### 3.9.5. Ask the monitoring system

The deployment includes prometheus to collect metrics from k8s and the running applications. To access the metrics, a grafana instance with many default dashboards is deployed as well. To access the dashboards, an ingress route “edp-portal.api.*. <customer>.net/grafana” is created.

Grafana also uses keycloak as authentication server, the user “admin” should have all necessary permissions.

When logged-in to Grafana and clicking on Dashboards > Manage a list of available dashboards as depicted in the following image is available.

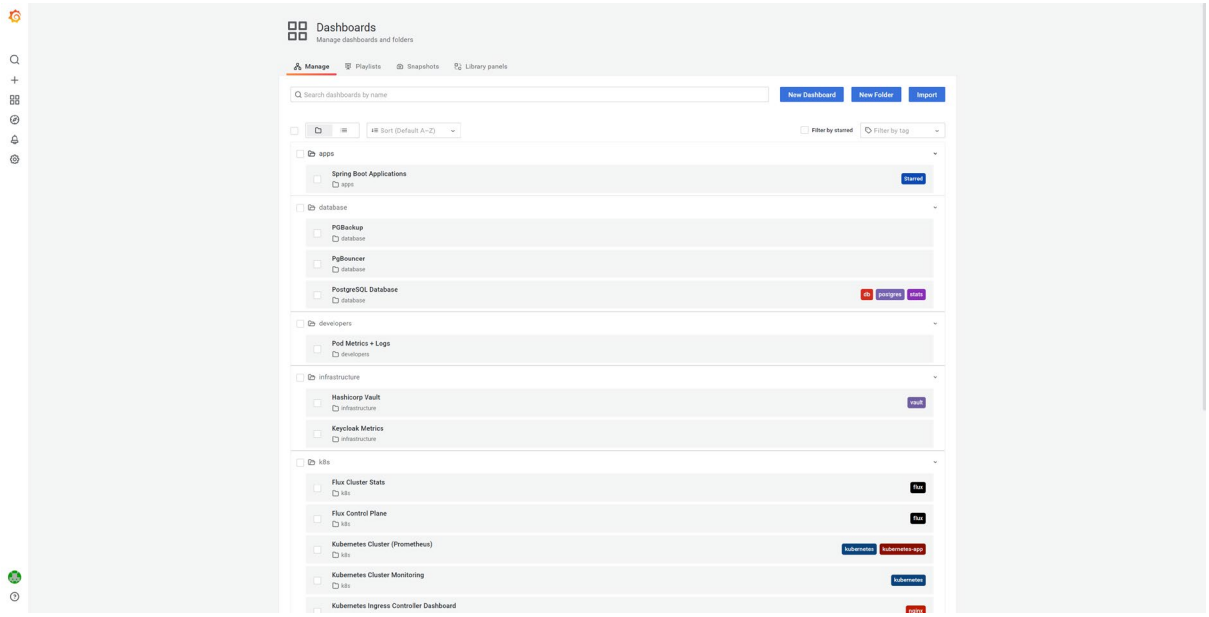



*Fig. 4 All available Grafana monitoring dashboards*

The following dashboards are available:

| Path | Dashboard Name | Brief description |
|---|---|---|
| apps | Spring Boot Applications | This dashboard shows details about any spring boot application, especially JVM-related metrics. |
| developers | Pod Metrics + Logs | This dashboard shows a quick overview of a Pod's CPU, Memory and Network usage and its logs. |
| infrastructure | Hashicorp Vault | This dashboard shows details of Vault operations. |
| infrastructure | Keycloak Metrics | This dashboard shows an overview of Keycloak operations. |
| k8s | Flux Cluster Stats | This dashboard shows an overview of Flux operations. |
| k8s | Flux Control Plane | This dashboard shows details about Flux operations. |
| k8s | Kubernetes Cluster (Prometheus) | This dashboard shows a short overview of the entire Kubernetes cluster. |
| k8s | Kubernetes Cluster Monitoring | This dashboard shows a detailed overview of the entire kubernetes cluster. |
| k8s | Kubernetes Ingress Controller Dashboard | This dashboard shows the same data as the NGINX Ingress controller dashboard. |
| k8s | Kyverno | This dashboard shows details about Kyverno operations. |
| k8s | NGINX Ingress controller | This dashboard shows details about all routes in the NGINX ingress controller. It can be used to get an overview of which routes have problems. |
| k8s | Prometheus 2.0 Overview | This dashboard shows details about Prometheus operations. |
| monitoring | Alerts | This dashboard is an overview over the active Alerts in the cluster. |
| monitoring | Loki | This dashboard shows details about Loki operations. |
| General | Grafana Internals | This dashboard shows details about Grafana operations. |
| General | State of the Deployment | This dashboard shows an overview of the current state of the Flux deployment, including when the last reconciliation happened. |

# 4. E3 Server Deployment

The E3 server (=Edge-Device External Endpoint ) is a dedicated virtual host that is responsible for communication with Edge-Devices via the internet. It needs direct access to the Load Balancer hosting the Ingresses of the EDP cluster in order to proxy the requests to the EDP services.

This virtual host is therefore running inside the same Alicloud virtual private cloud (VPC) as the k8s cluster. Each environment has a dedicated E3 host that Edge-Devices are sending their requests to. Edge-Devices can be switched from one environment to the other.

The following components are running on the E3 server:

- **Apache2:** request authentication, reverse proxying
- **Notary:** docker image signature verification
- **Docker registry:** hosts docker images for Edge-Devices
- **Debian Packages:** Debian OS package registry for Edge-Devices
- **Git:** Edge-Device repositories, machine repositories

## 4.1. Requirements

In order to be able to install a new E3 server you need a separate Linux-based deployment host from which you can reach your prospective E3 server. This deployment host needs to have the **Ansible 2.8+** installed with following additional collections:

- ansible.posix
- community.general
- community.crypto
- community.docker

The deployment is performed via an ansible script that needs to be cloned to the deployment host from https://bitbucket.org/kmooon/edp-e3.git

This script will install an E3 server on any external target host with fresh **CentOS 8** installation.

During installation the script needs access to the already installed and configured **Vault** inside the K8s cluster.

## 4.2. DNS

Configure the DNS server so the host can reach the following subdomains in your selected domain (e.g. testing.api. <customer>.net).

- edp-docker
- edp-git
- edp-monitoring

- edp-onboarding
- edp-packages
- edp-access

## 4.3. Firewall

Configure your firewall so the following ports are open:

| Port | Used by |
|---|---|
| **443** | Backend services for Edge-Devices |
| **80** | Let's Encrypt SSL validation |
| **2222** | SSH maintenance access |
| **9100** | Prometheus node exporter |

Make sure the E3 can reach the following EDP services in the K8s cluster:

- onboarding-service (curl should return 204)
  ```
  curl https://edp-service.api.testing.<customer>.internal/onboarding/health -i
  ```
- monitoring-service (curl should return 202)
  ```
  curl -X POST https://edp-
  service.api.testing.<customer>.internal/monitoring/api/v1/monitoring/container -i -d {}
  ```
- vault (curl should return 200)
  ```
  curl https://edp-service.api.testing.<customer>.internal/vault/v1/sys/health -i
  ```

If you use a firewall for outgoing traffic, ensure the Debian packages master repository is reachable via 22/tcp (SSH).

## 4.4. Custom TLS trust

This step is only required if the host certificate(s) of your EDP services is signed by a custom CA which is not well known.

The steps are to be executed on the target host.

1. Copy your CA chain to /etc/pki/ca-trust/source/anchors/
2. Run
   ```
   update-ca-trust extract
   ```

## 4.5. Docker Installation

1. Install and start docker
   ```
   dnf config-manager --add-repo=https://download.docker.com/linux/centos/docker-ce.repo
   dnf install -y docker-ce
   systemctl enable --now docker
   ```
2. Login to Cloud Container Registry (with service user)
   ```
   docker login  <customer>-net-ecr-registry-vpc.eu-central-1.cr.aliyuncs.com
   ```
3. The following images will be fetched by docker from Cloud during the ansible playbook run (see chapter 3.6.2)

- Apache2:
  <customer>-net-ecr-registry-vpc.eu-central-1.cr.aliyuncs.com/<customer>-edp-deployment/httpd-git
- Maintenance-Access:
  <customer>-net-ecr-registry-vpc.eu-central-1.cr.aliyuncs.com/<customer>-edp-deployment/ssh-maintenance-access

**Warning:** The password will be stored unencrypted in */root/.docker/config.json*. Do not use your personal credentials!

## 4.6. E3 Deployment

### 4.6.1. Configuration setup

The following steps are performed on your deployment host in the ansible/ subdirectory of your cloned edp-e3 repository. These steps have already been done for the E3 deployment. You only need to modify the playbook or hostfiles if you need to change a configuration. It is recommended to clarify your changes with DAIM before deploying changes to E3 as this may have a negative effect on the Edge-Device processing.

1. Create a credential file vars/<HOST>-credentials.yml containing the following 3 entries (see next chapter for valid **HOST** names, **YOUR_VAULT_TOKEN** from vault-secret in Cluster, bitbucket **TOKEN** from your bitbucket profile management https://id.atlassian.com/manage-profile/security/api-tokens):
   ```
   # vars/<HOST>-credentials.yml
   vault_root_token: YOUR_VAULT_TOKEN
   git_pull_credential: https://user:TOKEN@bitbucket.org
   device_repository_upstream: https://bitbucket.org/kmooon/edp-provisioning-master.git
   ```
2. Copy an existing e3 playbook and modify it for your needs (at least hosts and the path to the included credential file)
3. Copy an existing appropriate variable file in host_vars/ and modify it for your needs
4. Add your host to the inventory file to the suitable groups (at least to group e3).
5. Check that ansible can reach the target hosts
   ```
   ansible -m setup -u root HOSTNAME
   ```

Hint: You can fetch the vault token from the k8s cluster using:

```
kubectl get secret -n vault | grep vault-token
kubectl get secret -n vault vault-token-XXXX -o=json | jq -r '.data.root_token' | base64 -d
```

### 4.6.2. Deployment

Install common environment and tools to the E3 host. This is not required but highly recommended.

```
ansible-playbook common.yml -l HOSTNAME
```

Deploy the server components for E3 connected to the Cluster:

```
ansible-playbook e3-alibaba.yml -l HOSTNAME
```

The host-specific settings are available in the folder **ansible/host_vars**. The following hosts are relevant:

- e3-<customer>-testing
- e3-<customer>-staging
- e3-<customer>-production

## 4.7. Post-deployment steps

The following steps must be done after the initial E3 deployment.

### 4.7.1. Package synchronization setup

1. Read the ssh public key from **packages** user:
   ```
   cat /home/packages/.ssh/id_ed25519.pub
   ssh-ed25519 AA… packages@your-e3-server
   ```
2. Send the SSH key to DAIM hopferwieser@unisoftwareplus.com so it can be added to the Debian master repository server (edge.unisoftware.plus)
3. Once the key is added to the server by DAIM, run Debian package sync on your E3 host
   ```
   su - packages
   sync-packages
   ```

### 4.7.2. Known Issues

None

## 4.8. Error connecting container to network backend_network

The deployment fails with following message: *Error connecting container to network backend_network*

A restart of the docker daemon should fix the problem:

```
systemctl restart docker
```

# 5. Edge-Device

Physical Edge-Devices are usually provisioned directly by your hardware supplier during the production process via the **DAIM install server**.

This install server provides an automated installation procedure via ethernet and PXE boot. The install server is always customized and contains the customer specific “configuration UI” and onboarding endpoints that the Edge-Devices need to contact for device onboardind. The onboarding endpoint look like:

- https://edp-git.api.production.<customer>.net
- https://edp-onboarding.api.production.<customer>.net

## 5.1. Provisioning of an Edge-Device from PC

The Install Server is provided by DAIM as VDI image, which can be booted on any PC with VirtualBox.

In this scenario the Edge-Device is directly connected to a laptop or PC (here called host) with an ethernet cable and should be installed via PXE. The install server should run as VM via VirtualBox (commands for a Linux host machine).

1. Configure the network interface on the host to use IP address 192.168.100.1/24.
   ```
   ip addr add 192.168.100.1/24 dev enp0s31f6
   ip link set enp0s31f6 up
   ```
2. Configure the first VM network adapter (Settings -> Network) as bridged
3. Start and login to VM
4. Configure the network interface on VM to use IP address 192.168.100.2/24.
   ```
   ip addr add 192.168.100.2/24 dev eth0
   ip route add default via 192.168.100.1
   ```
   Note: The route command is important, otherwise the the perl module Net::Address::IP::Local in /usr/local/sbin/filter-serverip.pl cannot detect the public IP address and the IP replacing in the installer scripts will not work.
5. Start Dnsmasq
   ```
   systemctl start dnsmasq
   ```
6. On startup of the Edge-Device, go to system BIOS select “PXE Boot”. In the boot menu select your desired target endpoint:
   a. Production
   b. Staging
   c. Testing
7. In next submenu, select desired release version
   a. Release 0.5.9 (recommended)
   b. Master
   c. Develop
8. In next submenu, select installation variant
   a. Development
   b. Customer/Production

## 5.2. Remote access to an Edge-Device (in developer mode only)

Please note, that by default any terminal access is disabled on Edge-Devices for security reasons! Only Edge-Devices that are set up in **developer mode** can be accessed via secure shell (SSH).

DAIM will provide an SSH maintenance access service for production devices in a future release of the Platform. This will allow access to any Edge-Device via a reverse SSH tunnel that is created from the device to the platform backend services.

A how-to will be added to this document when the service is available.

## 5.3. Edge-Device disk storage

An Edge-Device holds a 128 GB Industrial Grade SSD. The disk is partitioned in a 47 GB system partition which holds the operating system and the docker environment, and a second partition used for LVM snapshots and for the upgrading mechanism.

On the system partition, around 5 GB are used by the operating system and the DAIM services. Note that this is only an estimated value, and it could be exceeded in the future if new space consuming features are integrated. A maximum of 4 GB is reserved for log files.

As a consequence, around 35 GB of space are left for Docker containers, images and volumes (3 GB are reserved for the operating system services).

**Please note that volumes are deleted during major upgrades (e.g. 0.3.x -> 0.4.x) for versions < 0.6.1**

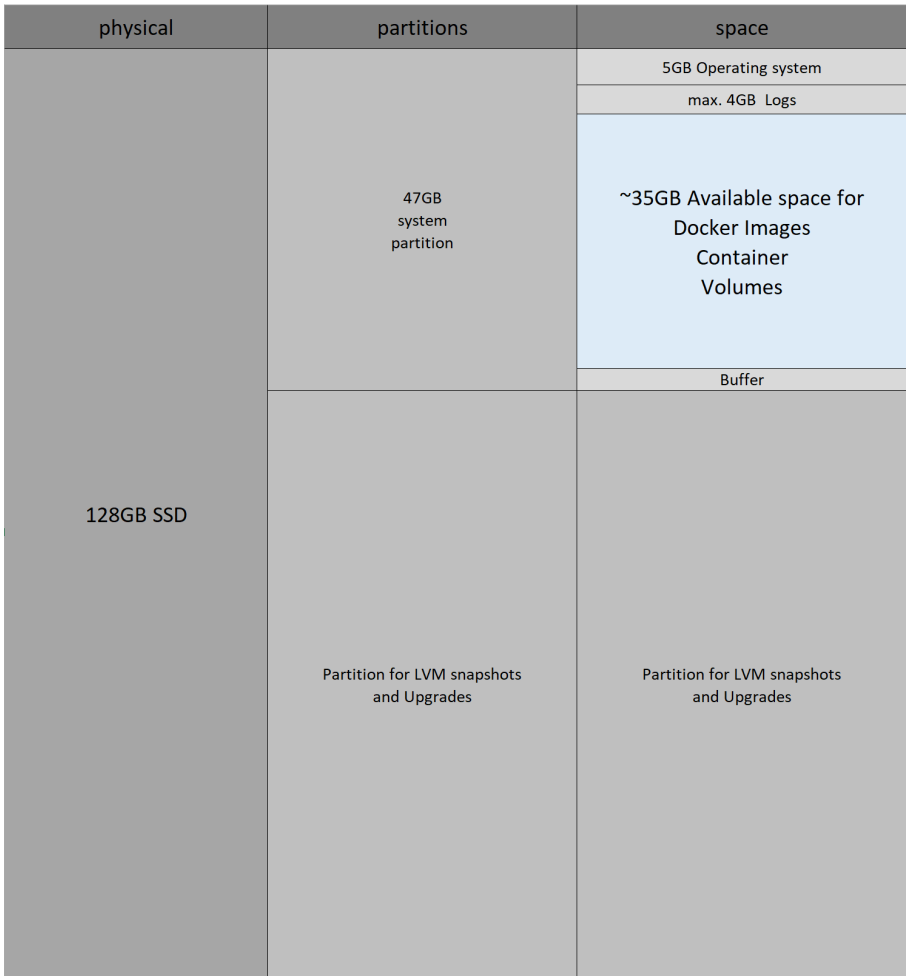



*Fig. 5 Illustration of the physical storage of an Edge-Device with 128GB HDD*

# 6. Change Log

| Date | Version | Change description |
|---|---|---|
| 24.05.2023 | Revision 14 | Changelog added; General rework |
| | | |
| | | |
| | | |