Container-Orchestrierung mit Kubernetes

FFG 2018 GUUG

March 1, 2018

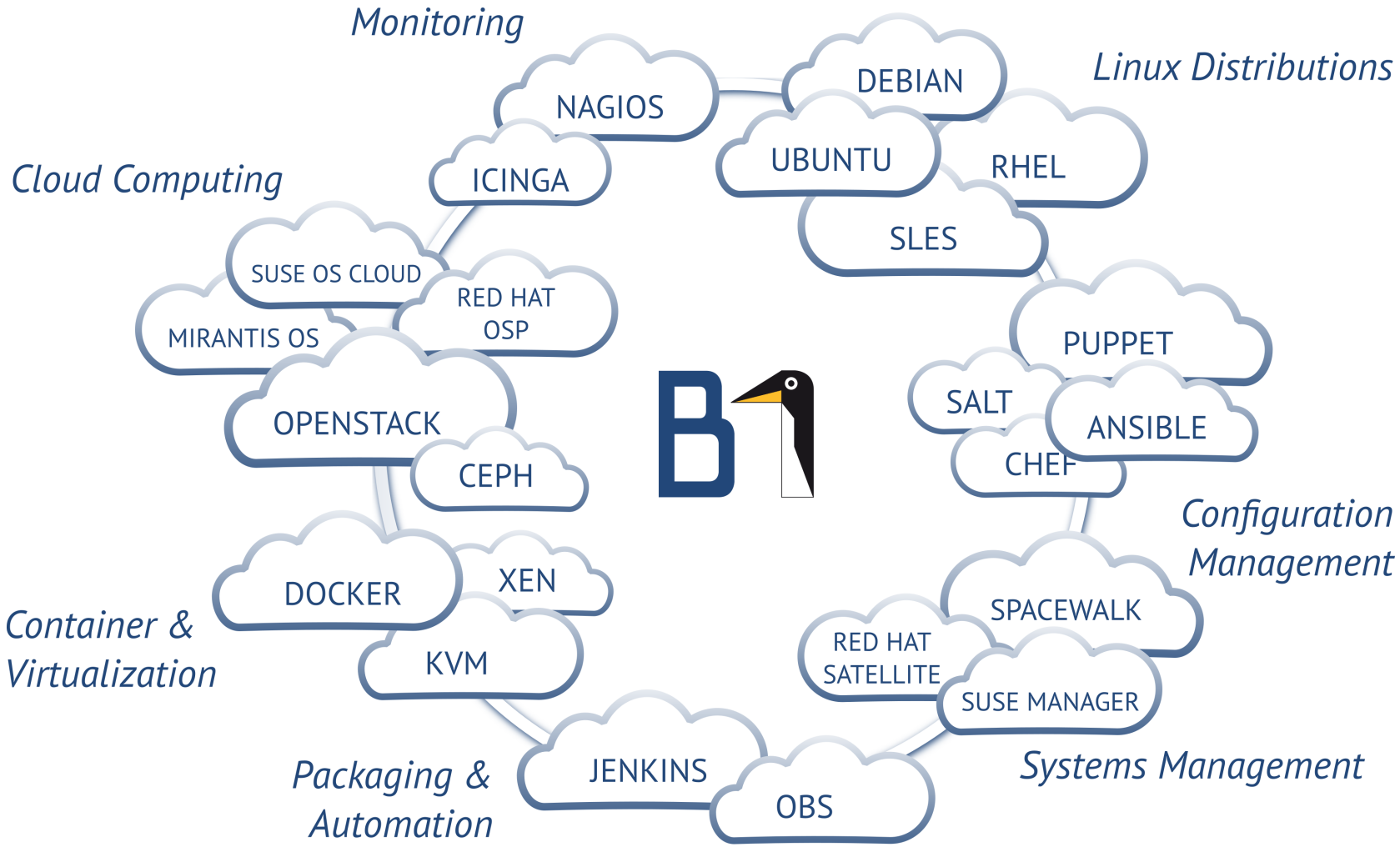
Michael Steinfurth
Linux/Unix Consultant & Trainer
B1 Systems GmbH
steinfurth@b1-systems.de

André Steincke
Linux Consultant & Trainer
B1 Systems GmbH
steincke@b1-systems.de

Introducing B1 Systems

- founded in 2004
- operating both nationally and internationally
- about 100 employees
- vendor-independent (hardware and software)
- focus:
  - consulting
  - support
  - development
  - training
  - operations
  - solutions
- offices in Rockolding, Berlin, Cologne & Dresden

Areas of Expertise



Workshop Setup

quick Host Setup

- three nodes
- recommended CentOS 7 (here SLES 12 SP3) with extra and updates repo
- make hostnames resolvable on all hosts (dns, hosts file)
- (optional) share ssh keys between all hosts

Software Components

- `etcd` cluster – key value store
- `flanneld` – overlay network
- `docker` – container engine
- `kubernetes` packages

`etcd` Cluster

- necessary for Kubernetes, `flanneld`
- part of the coreos project
- distributed key value store
- high availability is necessary
- quorum majority needed

↪ run on odd number of DCs (minimum 3)

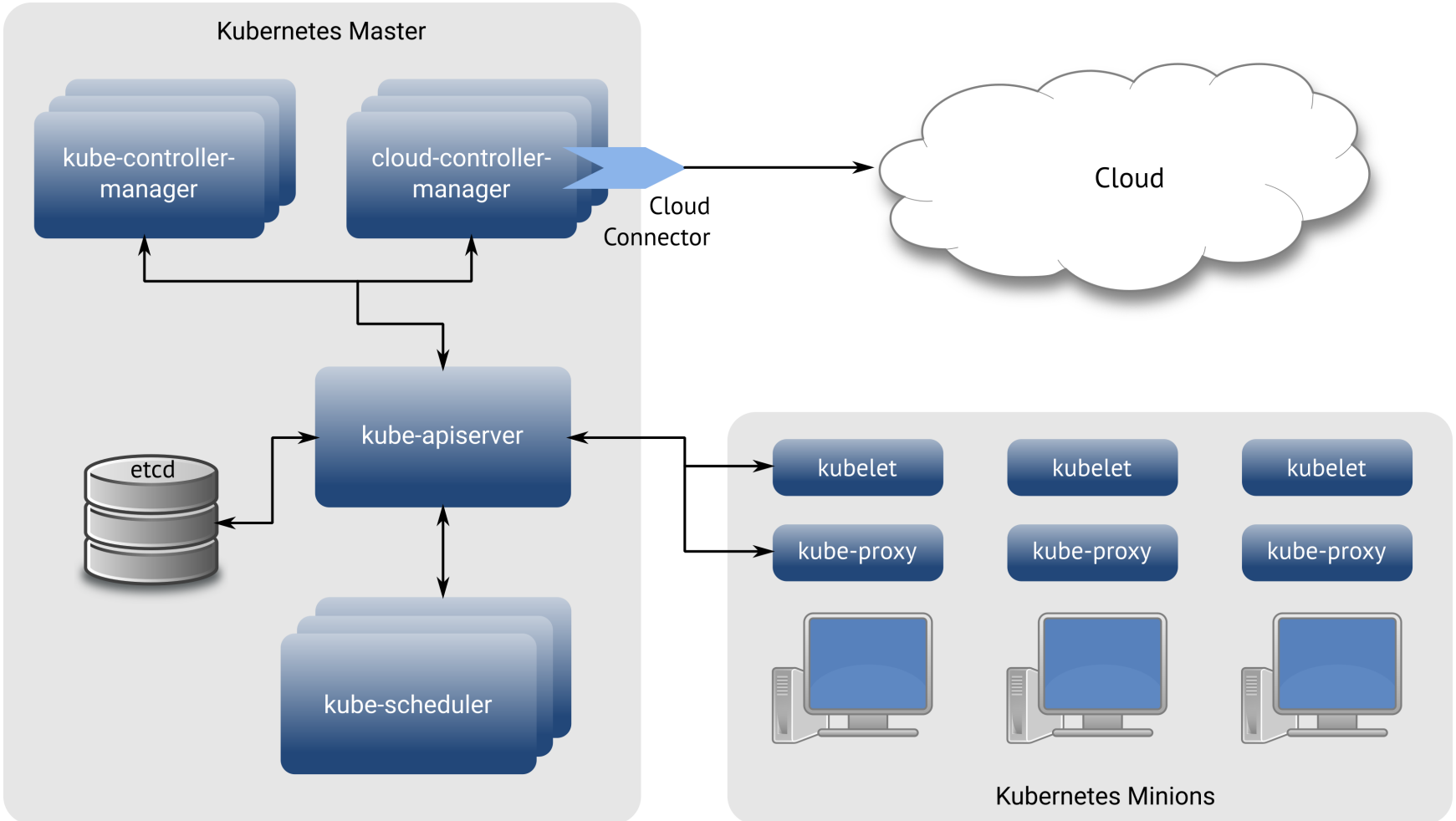
Overlay Network – `flanneld`

- containers need to communicate on multiple nodes
- multiple solutions
  - `flannel`, `weave` plugin, `opencontrail`
- `flannel` integrates most easily + quite good performance
- part of the coreos project
- `flanneld` on each worker node

Kubernetes

- Kubernetes master and worker
- master component = management plane
  - schedules, supervises replicates pods
  - saves configuration data, manages volumes
  - API interface
- worker component connects to container backend
  - creates *real* container
  - creates service connections (`iptables`)
  - container image handling

Kubernetes Architecture



Source: https://kubernetes.io/docs/concepts/architecture/cloud-controller/

Installation & Configuration

Packages to Install

Install packages, run command on each node

```
# zypper install etcd etcdctl flannel \
kubernetes-master kubernetes-node kubernetes-client
```

etc configuration on first node (adapt HOSTNAME: *vm1*)

```
# [member]
ETCD_NAME="vm1"
ETCD_LISTEN_PEER_URLS="http://0.0.0.0:2380"
ETCD_LISTEN_CLIENT_URLS="http://0.0.0.0:2379"
ETCD_INITIAL_CLUSTER="vm1=http://vm1:2380"
ETCD_INITIAL_CLUSTER_STATE="new"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_ADVERTISE_CLIENT_URLS="http://vm1:2379"
```

etcd Cluster Start and Check

1. Start `etcd` on each node quickly one after the other:

```
# systemctl start etcd
```

2. Check cluster state on one node:

```
# etcdctl cluster-health
```

flannel Configuration

- Set basic flannel configuration as key in etcd:

```
# etcdctl set /network/config \
  '{ "Network": "172.19.0.0/16", \
  "Backend": { "type": "host-gw"} }'
```

- Edit /etc/sysconfig/flanneld on each node (adapt hostnames):

```
FLANNEL_ETCD_ENDPOINTS="http://vm1:2379"
FLANNEL_OPTIONS="--etcd-prefix=/network"
```

`flanneld` Start and Check

1. Start `flanneld` on each node:

```
# systemctl start flanneld
```

2. Check if `flanneld` is running:

```
# systemctl is-active flanneld
# ip route show
```

Edit /etc/kubernetes/apiserver on vm1

```
KUBE_API_ADDRESS="--insecure-bind-address=0.0.0.0"
KUBE_ETCD_SERVERS="--etcd-servers=http://vm1:2379"
KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=172.20.0.0/16"

#just remove "ServiceAccount"
KUBE_ADMISSION_CONTROL="--admission-control=NamespaceLifecycle,LimitRanger,DefaultStorageClass,\
ResourceQuota"
```

Kubernetes Master scheduler + controller-manager

/etc/kubernetes/controller-manager

```
KUBE_CONTROLLER_MANAGER_ARGS="--cluster-name=kubernetes"
```

/etc/kubernetes/scheduler

```
KUBE_SCHEDULER_ARGS=""
```

Kubernetes Master Start and Check

1. Start master components on `vm1`:

```
# systemctl start kube-apiserver kube-controller-manager\
  kube-scheduler
```

2. Check if all are running on `vm11`:

```
# systemctl is-active "kube*"
# kubectl cluster-info
```

Kubernetes Worker

/etc/kubernetes/kubelet – on each worker node

```
@@ -2,13 +2,13 @@
KUBELET_ADDRESS="--address=0.0.0.0"
#KUBELET_HOSTNAME="--hostname-override=127.0.0.1"

 # Add your own!
KUBELET_ARGS="--pod-manifest-path=/etc/kubernetes/manifests\
--fail-swap-on=false\
--kubeconfig=/var/lib/kubernetes/kubeconfig"
```

/etc/kubernetes/config – on each worker node

```
KUBE_MASTER="--master=http://vm1:8080""
```

/etc/kubernetes/proxy – on each worker node

```
KUBE_PROXY_ARGS="--kubeconfig=/var/lib/kubernetes/kubeconfig"
```

Kubernetes Worker

/var/lib/kubernetes/kubeconfig – on each worker node

```
apiVersion: v1
kind: Config
clusters:
  - cluster:
      server: http://vm1:8080
    name: kubernetes
contexts:
  - context:
      cluster: kubernetes
      user: kubelet
    name: kubelet-to-kubernetes
current-context: kubelet-to-kubernetes
users:
  - name: kubelet
```

Kubernetes Worker Start and Check

1. Start master components on `vm2` , `vm3` ,`vm1` (optional):

```
# systemctl start kubelet kube-proxy
```

2. Check if both are running on `vm2`, `vm3`, `vm1` (optional):

```
# systemctl is-active "kube*"
```

3. Get cluster nodes' states, execute on `vm1`:

```
# kubectl get nodes
```

Kubernetes Master `apiserver`

1. Remove service account
2. NodePort range for external access ports to applications ports
3. Set `service-cluster-ip-range`
4. (Optional) Add `etcd` cluster

Kubernetes cli – `kubectl`

kubectl

- configuration tool
- manage and maintain kubernetes clusters
- talks to api server
- default config file in `/.kube/config`
- comes with shell completion

```
# kubectl completion __SHELL__
```

- e.g. bash completion

```
# source <(kubectl completion bash)
```

kubectl Examples 1

- List and get detailed information about nodes in cluster:

```
# kubectl get nodes
# kubectl describe nodes
```

- Make node unschedulable (+ draining pods) and schedulable:

```
# kubectl cordon nodes
# kubectl drain nodes
# kubectl uncordon nodes
```

kubectl Examples 2

- Manage entities and applications:

```
# kubectl get (pods|replicasets|service) (-o wide|yaml) (-w)
# kubectl create -f __ENTITY.YAML__
# kubectl delete (pod|replicaset|service) __ENTITYNAME__
```

- Use for debugging of platform and applications:

```
# kubectl get events
# kubectl describe node (__NODE1__)
# kubectl logs (-f) __POD_NAME__
```

Applications with Kubernetes

Pods 1

- smallest entity
- one IP address
- consists of min. 2 containers
  - 1 infra container image (`gcr.io/google_containers/pause-amd64:3.0`, keeps network namespace)
  - min. 1 app container (e.g. `nginx`)
- other entities can control pods by matching labels
- stateless way of thinking:
  1. delete
  2. restart

  ↪ work the same way as before (different parameters: ip address, hostname)

Create a file pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: nginx
  name: mynginx
  namespace: default
spec:
  containers:
  - name: nginxcont
    image: nginx:latest
    imagePullPolicy: Always
    env:
    - name: LOGDIR
      value: /srv/logs
```

Working with Pods

- Start the pod:

```
# kubectl create -f pod.yaml
```

- Get information about pods:

```
# kubectl get pods
# kubectl get pods -o wide
# kubectl get pods mynginx -o yaml
```

Replicasets/Replication Controller

- controlling entity
- keeps track of running (number of) pods
- find pods to control by a matching label

  `run = nginx`

  `stage = test`
- Replicaset same as ReplicationController
- Replicaset additonally supports `matchLabels` selector out of given quantities `stage in (dev, test)`
- keyword `replicas` defines number of running pods

Replicasets – Label Matching

replicaset yaml

```
...
spec:
  selector:
    matchLabels:
      run: nginx
...
```

pod yaml

```
...
metadata:
  labels:
    run: nginx
...
```

Deployments

- special entity to control versioning of other entities (e.g. replicasets) and your applications
- based on used images you can update, downgrades
- supports different update strategies:
  - rolling update – "step by step" e.g.:
    1. stop one pod with old application version
    2. start one pod with new application version
    3. stop next pod with old application version
    4. start ...
  - recreate update
    1. stop all pods with old application version
    2. start all pods with new application version
- undo/redo operations supported

depl.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 1
  strategy:
    type: RollingUpdate
  template:
    spec:
      containers:
      - name: nginxpod
        image: nginx
        imagePullPolicy: Always
```

Deployment Inspection

- Create a deployment using the following command:

```
# kubectl create -f /home/tux/k8s/depl.yaml
```

- Get information about pods:

```
# kubectl get deployment -o yaml
# kubectl get replicaset
# kubectl get pods
```

- Name scheme:

```
|- Deployment:    <name>
|  |-Replica Set: <name>-<integervalue>
|           |-Pod: <name>-<integervalue>-<randomString>
```

Updates with Deployments

1. Change version of image in deployment yaml:

```
-         image: nginx:1.12
+         image: nginx:1.13
```

2. Apply new yaml to api:

```
# kubectl apply -f depl.yaml --record
```

Undo and History

1. List revision history:

```
# kubectl rollout history deployment nginx-deployment
```

2. Hint: compare revision count with number of replicasets:

```
# kubectl get rs
```

3. Undo operation by jumping to specific revision:

```
# kubectl rollout undo deployment nginx-deployment --to-revision=1
```

External Access to Applications

Services

- entities representing one application to other applications
- also the way to present your application to outside world
- uses combination of virtual IP addresses + port + transport protocol
- does not change on upgrade or downgrade of your apps
- hides internal addresses of applications
- supports loadbalancing to several pods of the same kind (same app)

Service Handling

Create the service

```
# kubectl create -f svc.yaml
```

Test and get information about service

```
# curl -vvv http://localhost:32222
# kubectl get svc -o yaml
# kubectl get ep
```

External service

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-svc
  namespace: default
spec:
  selector:
    run: nginxpod
  ports:
  - name: nginx-port
    port: 8080
    targetPort: 80
    protocol: TCP
    nodePort: 32222
  type: NodePort
```

External service

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-svc
  namespace: default
spec:
  selector:
    run: nginxpod
  ports:
  - name: nginx-port
    port: 8080
    targetPort: 80
    protocol: TCP
    nodePort: 33333
  type: NodePort
  sessionAffinity: ClientIP
```

Volumes, Configuration, Secrets

Volumes

- a way to save data outside the r/w-layer
- persistent
  - shared filesystem needed to save cluster-wide
  - external storage provider (`cephfs`, `netapp`) supported
  - e.g. `hostpath`, host-based mount, on each k8s worker same host path must exist
- temporary
  - volume lifetime = pod lifetime on a node
  - think of a typical Linux-like *tmp* directory
  - e.g. `emptydir`
- special volumes: `configmap`, `secret`

Volume Example hostpath

hostpath volume in pod yaml

```
spec:
  containers:
[...]
    volumeMounts:
    - mountPath: /etc/nginx/conf.d
      name: nginx-config-volume
  volumes:
  - hostPath:
      path: /mnt/configurationfiles
      name: nginx-config-volume
```

Configmaps

- get variable data into your pod
- cluster-wide, independent from other entities
- different way to create configuration data
- representing data in different ways in pods
  - file-based (via volumes)
  - environment variables (since v1.6)

Create configmap

```
# kubectl create configmap dbconfig --from-file=dbconfig.properties
```

Use configmap as volume in pod

```
spec:
  containers:
  - name: nginxcont
    [...]
    volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: dbconfig
```

Configmap with evironment variables

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: dbconfig-env
data:
  SERVER: db.example.org
  USERNAME: dbuser
  DATABASE: test
```

Usage in pod

```
  Containers:
  [...]
    envFrom:
    - configMapRef:
       name: dbconfig-env
```

Secrets

- centralized way to save sensitive data as passwords, tokens, keys
- not in cleartext, but base64
- more flexible way of saving secure information compared to a pod
- similar to configmaps
  - key/value from files or standard input
- data presented as
  - file (file name = KEY, content = VALUE)
  - ENV variable (name = userdef., content = value)

Create secret from file

```
# kubectl create secret generic secret1 --from-file=password
```

Use as volume in pod

```
spec:
  containers:
  - name: nginxcont
    [...]
        volumeMounts:
      - name: secretvol
        mountPath: /etc/config/access
  volumes:
    - name: secretvol
      secret:
        secretName: secret1
```

Secret Presented as Environment Variable in Pod

Use as volume in pod

```
containers:
[...]
  env:
   - name: PASSWORD
     valueFrom:
       secretKeyRef:
         name: secret1
         key: password
```

Advanced Features of Kubernetes

Daemonsets

- special entity
- run a pod on each Kubernetes worker node
- Examples:
  - logging services which need to read logs from each node
  - K8s master/additional components (`apiserver`, `scheduler`, `flannel`)

Example daemonset yaml file 1/2

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: fluentd-daemonset
  namespace: kube-system
spec:
  template:
    metadata:
      labels:
        run: fluentd
        env: tst
[...]
```

Example daemonset yaml file 2/2

```
[...]
      name: fluentd
    spec:
      restartPolicy: Always
      containers:
      - name: fluentd
        image: gcr.io/google_containers/fluentd-elasticsearch:1.19
        env:
        - name: FLUENTD_ARGS
          value: -qq
```

Thank You!

For more information, refer to info@b1-systems.de
or +49 (0)8457 - 931096