Continuous Delivery of Micro Applications with Jenkins, Docker & Kubernetes at Apollo

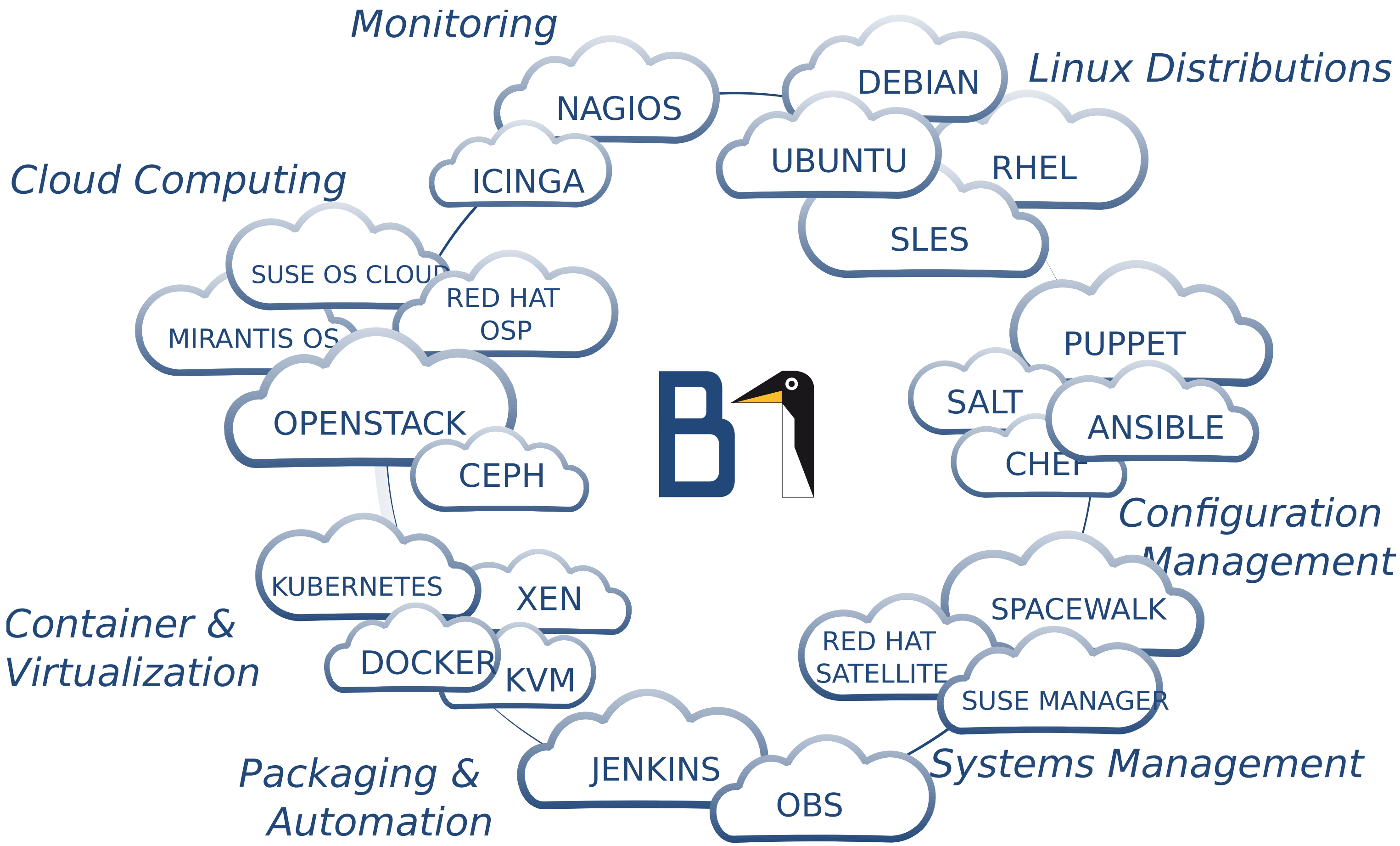
Ulrich Häberlein
Team Manager Backend Systems
Apollo-Optik Holding GmbH & Co KG

Michael Steinfurth
Linux / Unix Consultant & Trainer
B1 Systems GmbH

Introducing B1 Systems

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

Areas of expertise



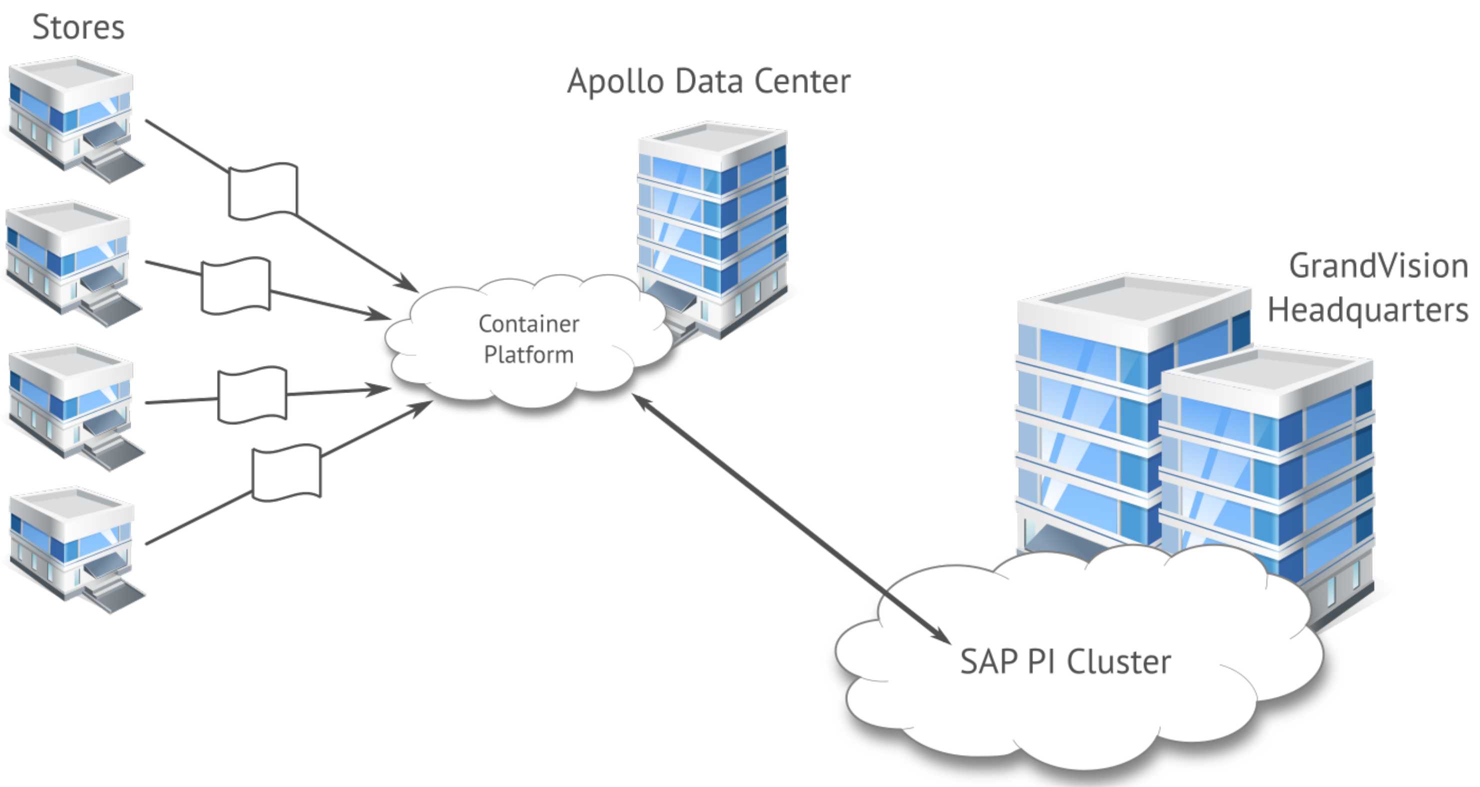
Introducing Apollo

- Germany’s largest optic retailer
  - founded in 1972
  - more than 800 stores in Germany
  - more than 100 stores in Austria

- part of grandvision, a global leader in optical retail
  - more than 6516 stores in 40+ countries
  - more than 31000 employees
  - more than 15 million spectacles

Business Case

Business case



Status quo

- legacy business platform with multiple databases
- 900 stores
- flat file interfaces provided by the POS database
- nightly batch processing of orders and master data updates
- centralized SAP business platform operated by GrandVision
- container-based middleware

Why run middleware with micro applications 1/2

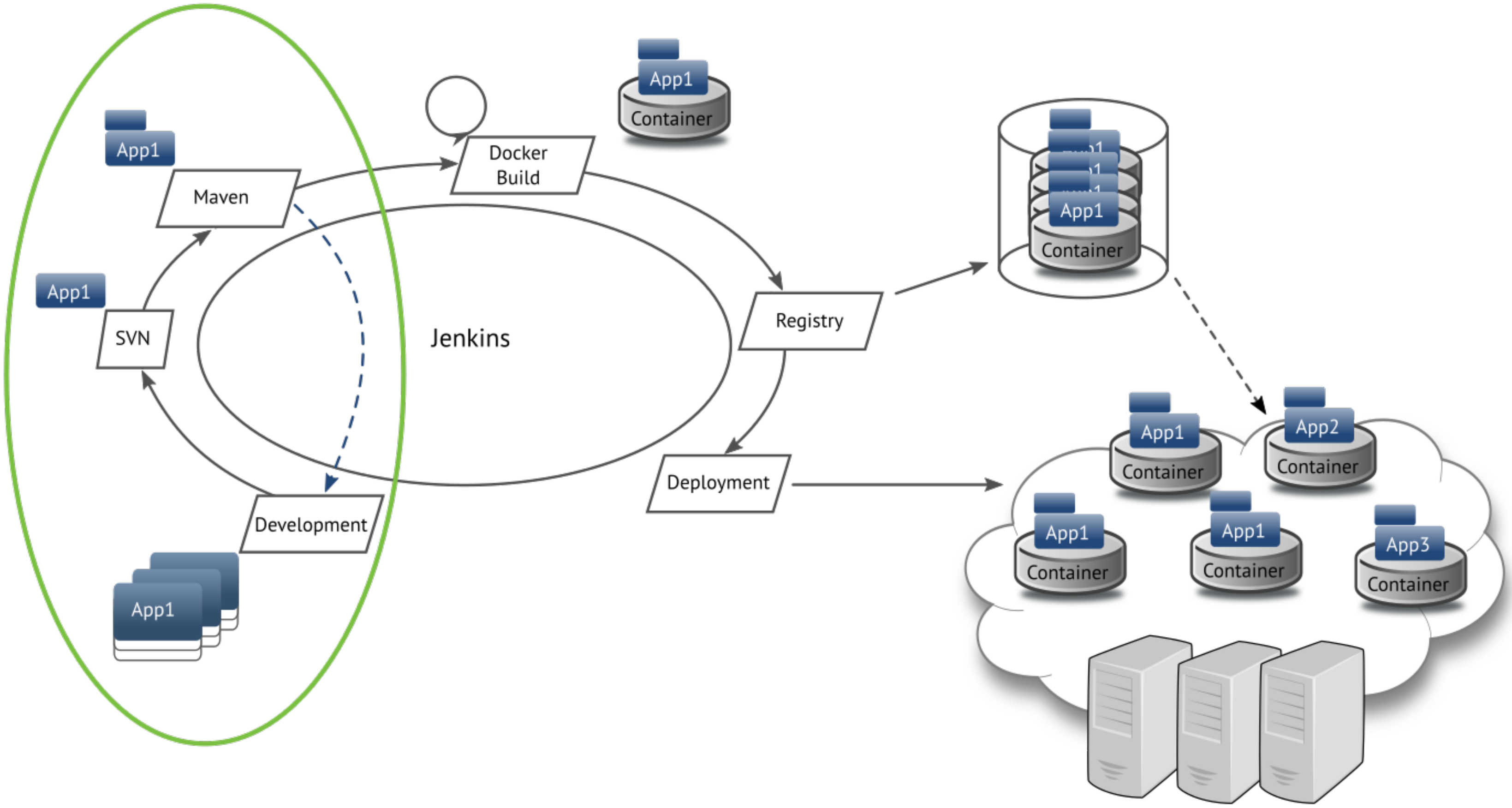
- agile development
- fast and changeable business processes
- easy to scale and expand
- continuous, automatic updates
- standardized test management

Why run middleware with micro applications 2/2

- guaranteed deployment quality
- high availability
- OS version independent
- configuration as code
- easy auditing with subversion

Workflow

Workflow



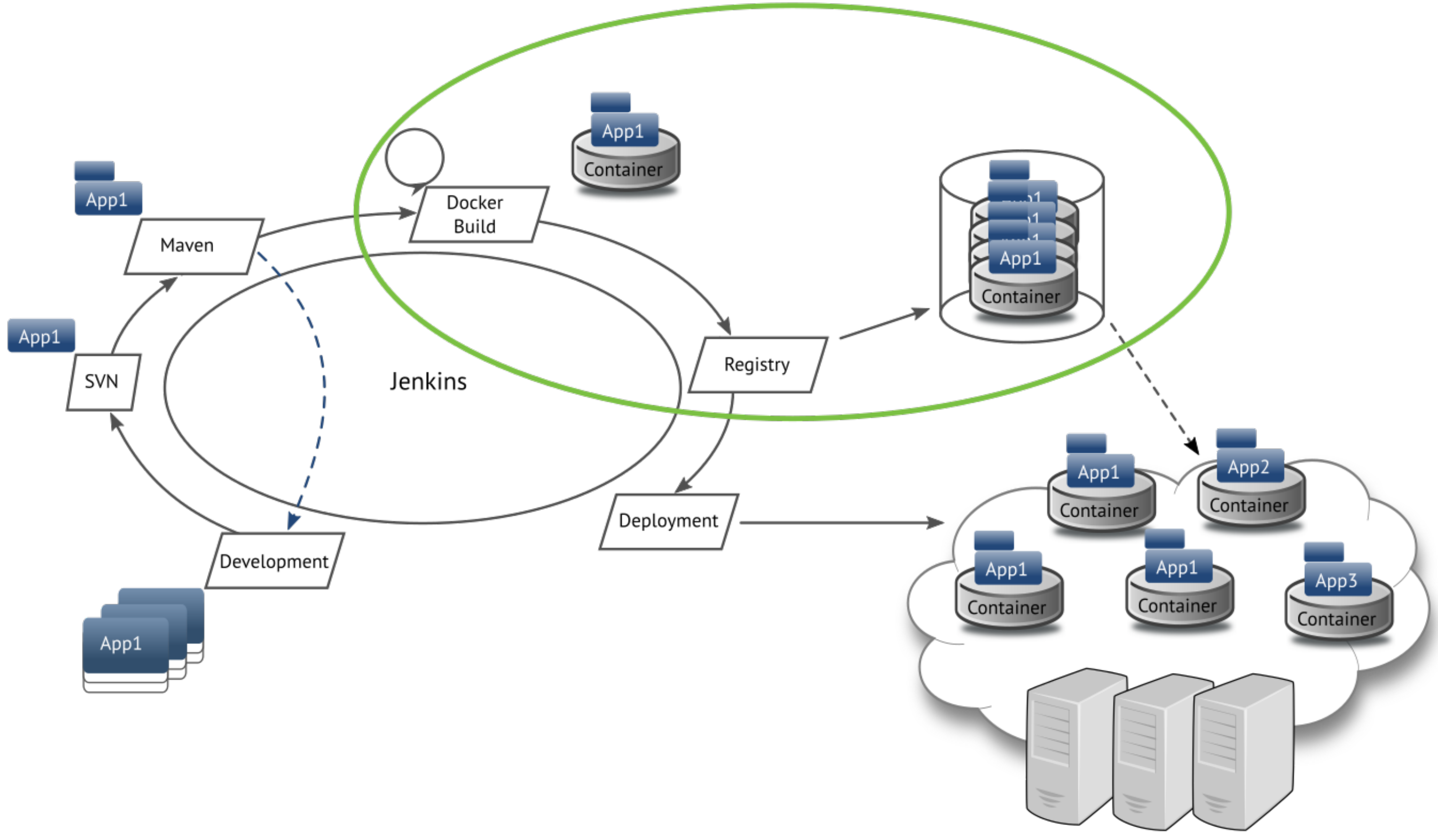
App1
Maven
Docker
Build
App1
Container
App1
Container
App1
SVN
Jenkins
Registry
Deployment
Development
App1
App1
Container
App2
Container
App1
Container
App1
Container
App3
Container

Jenkins processing loop

1) Java applications are developed and committed to SVN
2) Jenkins notices changes in SVN, handles the build job
3) maven → build jobs including dependency handling an testing
4) handover to micro app job

- developer work indepently no operation needed
- avg. job build duration: ~2 minutes (+initial cron offset)

Workflow



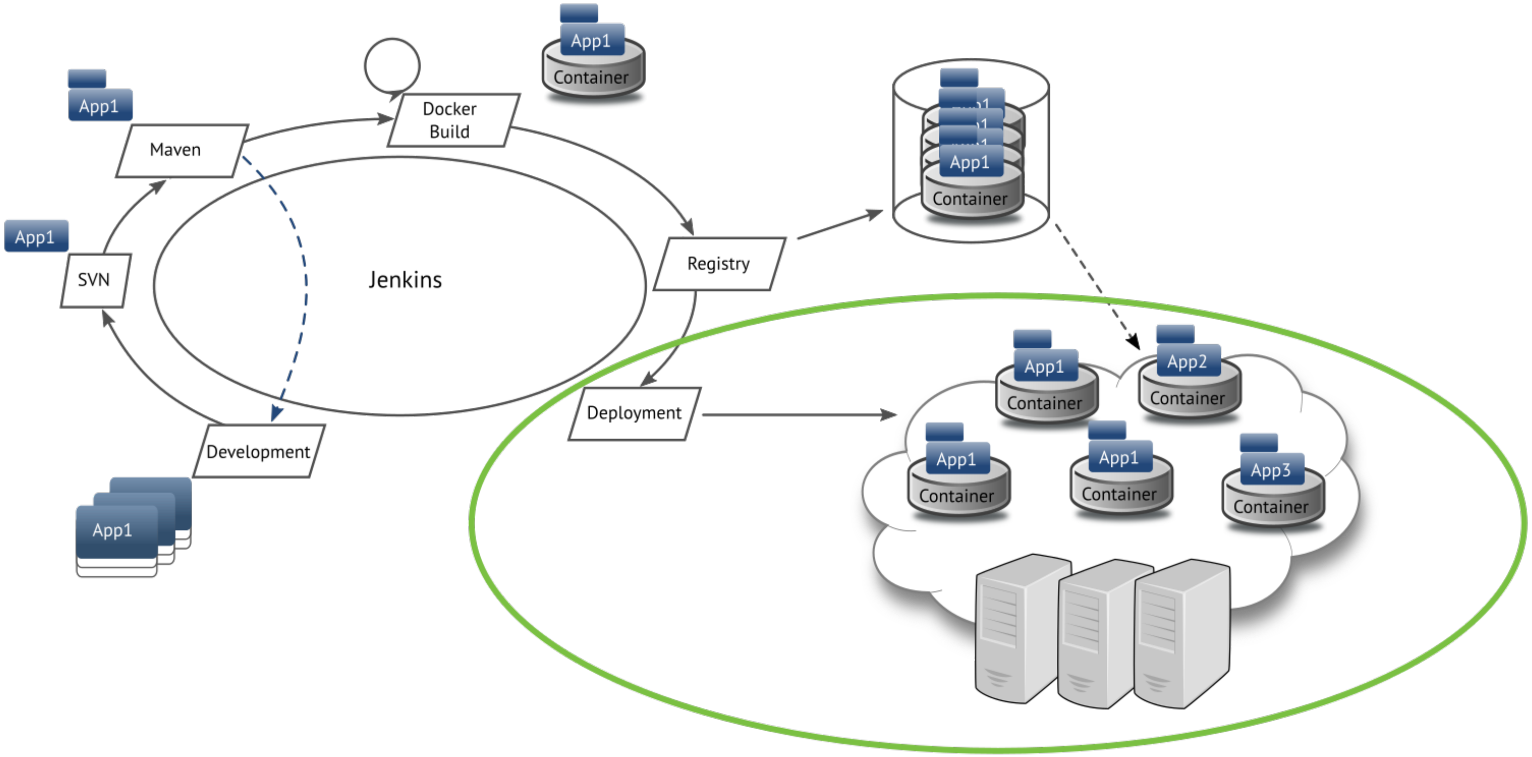
App1
Container
Docker
Build
App1
Maven
App1
Container
App1
SVN
Jenkins
Registry
App1
Container
App2
Container
Deployment
App1
Container
App1
Container
App3
Container
Development
App1

Build jenkins micro app container

1) package app into a generic container
2) populate configuration files
3) build Docker image
4) store meta information in Dockerfile (service ↔ ports)
5) push to registry

- completly automated
- avg. job build duration: ~1 minute

Workflow



App1
Container
Docker
Build
App1
Maven
App1
SVN
Jenkins
Registry
App1
Container
Deployment
Development
App1
App1
Container
App2
Container
App1
Container
App1
Container
App3
Container

Jenkins deployment

1) job handover

2) download image information from Registry (tag)

3) create meta information based on data from deployment NFS and Dockerfile

4) create NFS structure

5) adjust software state file (configuation NFS)

6) proceed with deployment job (direct deployment on devel stage)

- average job duration: 7 seconds

Example: deployment of the development stage

- Dockerfile labels + configuration information = YAML
  → Image-TAG and replication count given in central NFS configuration file
- direct deployment on Kubernetes platform for each micro application pod
- job is done by Jenkins slave
- average duration: 12 seconds

Deployment YAML (example)

```yaml
apiVersion: v1
kind: Deployment
metadata:
  name: testmicroapplication-deployment
spec:
  replicas: 1
  strategy:
    type: RollingUpdate
  template:
    spec:
      restartPolicy: Always
      containers:
      - name: testmicroapplication
      image: registry/apps/testmicroapplication:B123
      env:
```

Test staging

- software state file for development stage in svn repository
  → container name & version, min. & max. frequency, downtime value
- Jenkins job copies state file to test staging SVN repository
  → automatic discovery and restore check of all running pods (app containers)
  example state file:

```
testmicroapplication:B123:1:1:1
integration-sample:B42:2:3:1
```

Special temporary setup

- we broke the staging concept during development phase by doing this
- applications deployed immediately after testing and development
  → faster development progress, since feature development and bugfixing happen simultaneously
- total run time from commit to deploy: around 7 minutes

Infrastructure

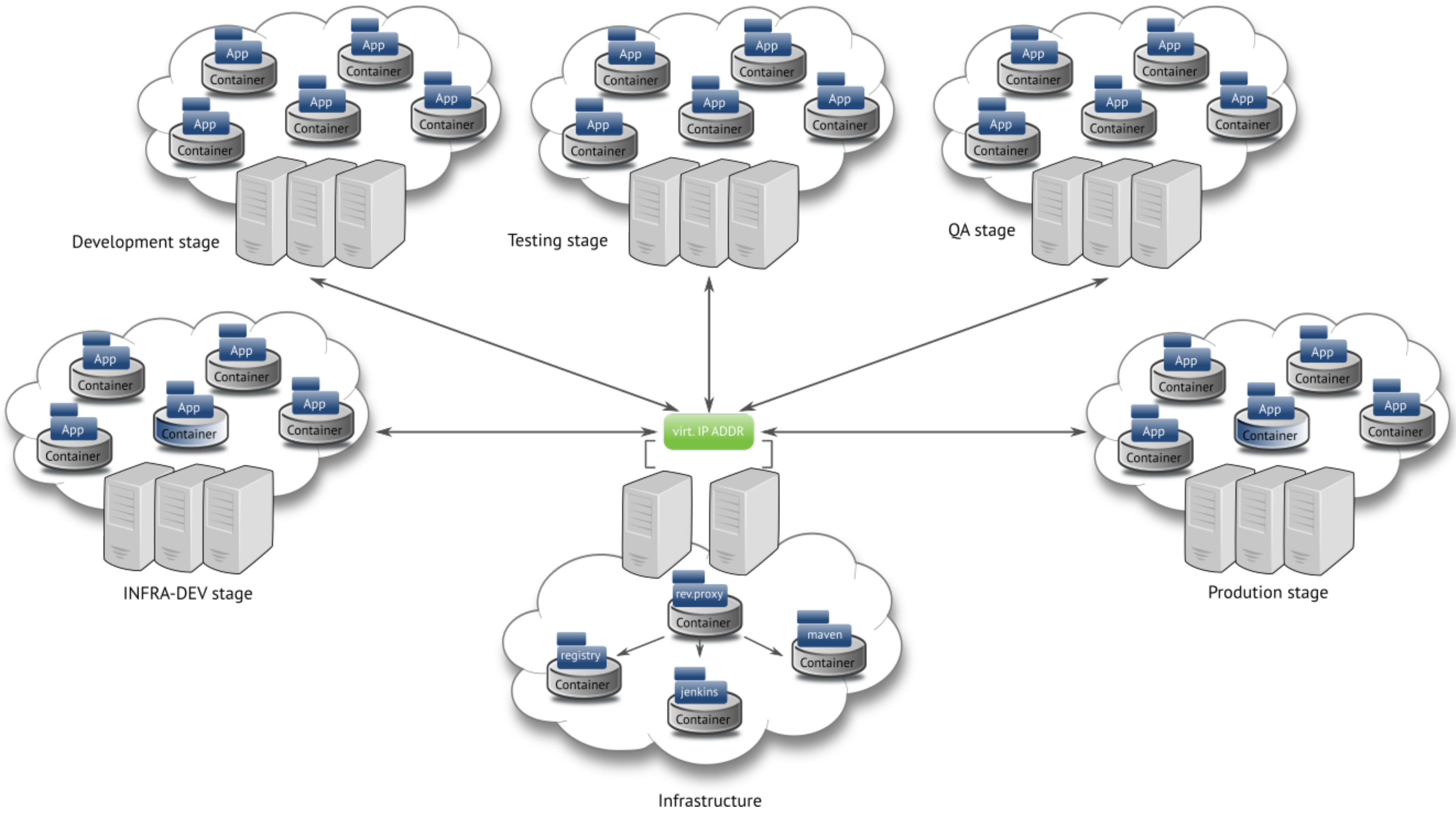
Host systems

- VMware virtual machines with SLES 12 SP2
- 3 VLANs for each staging area, each /16
- +1 VLAN cluster service IP address range
- one exclusive shared NFS volume for all hosts of one stage
- SUSE Manager deployment
- standardized systems → new systems configured and integrated in less than five minutes

Kubernetes

- virtual network using Flanneld
- Kubernetes packages provided with given SUSE Manager
- started at version 1.3 , now at 1.5
- repos synced from Open Build Service (master & worker)
- combined master and worker usage per node
  → higher availability with easy scalability
- Docker container backend
- outsourcing of infrastructural dependencies (avoid chicken-or-egg question)
  → components were not run as infrastructure pods themselves
  → separate registry, *etcd*-cluster

Infrastructure

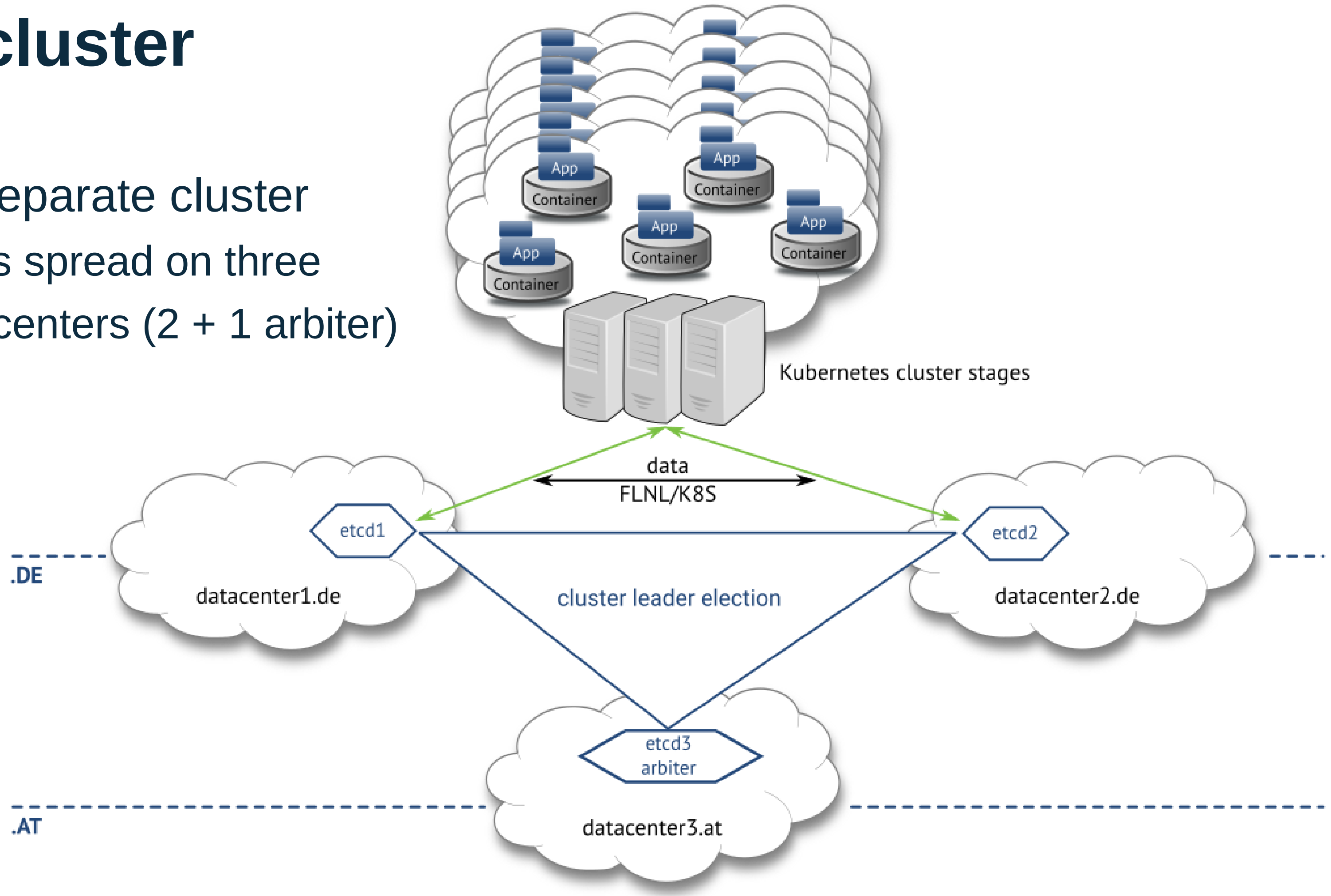


Infrastructure components

- *Registry*, *Jenkins*, *Maven* → higher availability through containerization
- reverse proxy with dynamic configuration for vhosts from Kubernetes services
- almost all components are build and deployed with Jenkins

Etcd cluster

- *etcd*: separate cluster
  - nodes spread on three data centers (2 + 1 arbiter)



Used Kubernetes features

- using rolling updates in replica sets via Kubernetes deployment entities
  → no application downtime
- use of the Kubernetes proxy with service entities to distribute network load on multiple application container
  → Iptables distributes traffic regardless of the incoming IP address
- horizontal pod autoscaler – *(experimental status)*
  → configurable scaling during times of performance bottlenecks

Logging

- logging prepared for *ELK* stack
- the stack in Kubernetes with distributed storage
- *fluentd* to capture container logs from Kubernetes host
- applications write log files themselves
- ongoing development of the applications for *ELK* stack connectivity

Service monitoring

- using given Nagios for basic system/service monitoring
- monitored:
  - Kubernetes daemons
  - Flannel, etcd
  - Docker (storage-driver btrfs free space)
  - Shared storage NFS free space

Performance graphing (planned)

- performance graphing for each cluster
- nodes and containers
- used time series data provided by heapster
- grafana + influxdb as backend on each cluster

Observations & lessons learned

Observations

- infrastructure faster than expected
- on peak load we are spreading the work of the Jenkins jobs
  → done by dynamically generated, temporary, Jenkins slaves running on the development Kubernetes platform itself
- environment very stable and pretty much self-healing

Lessons learned

- make etcd highly available without chicken-or-egg problem
  - ended at arbiter 3 DC spreaded cluster
- updating cluster of Rabbitmq message broker requires a lot of attention and manual interaction (Erlang version)

Thank you!

SUSE®

We adapt. You succeed.