

(/)

Compliant Kubernetes

(<https://elastisys.com/category/compliant-kubernetes/>),

Elastisys Engineering

(<https://elastisys.com/category/tech-post/engineering/>),

Technical posts (<https://elastisys.com/category/tech-post/>)

---



## Building and testing base images for Kubernetes cluster nodes with Packer, qemu and Chef InSpec

© Elastisys Tech Blog(<https://elastisys.com/author/elastisys-com/>)

February 13, 2020(<https://elastisys.com/2020/02/13/>) © 9:53 am



(http 

s://w (http    
ww.l s://n (http (http  
inke ews. s://t s://r  
din.c yco witt eddi  
om/c mbi er.c t.co  
ws/s nato om/i m/su  
hare r.co nten bmit  
? m/su t/tw ?  
url= bmit eet? url=  
http link? text http  
s://el u=h =Bu s://el  
astis ttps: ildin astis  
ys.c //ela g ys.c  
om/ stisy and om/  
buil s.co testi buil  
ding m/b ng ding  
- uildi base -  
and- ng- imag and-  
testi and- es testi  
ng- testi for ng-

base ng- Kub base  
- base erne -  
imag - tes imag  
es- imag clust es-  
for- es- er for-  
kube for- node kube  
rnet kube s rnet  
es- rnet with es-  
clust es- Pack clust  
er- clust er, er-  
node er- qem node  
s- node u s-  
with s- and with  
- with Chef -  
pack - InSp pack  
er- pack ec at er-  
qem er- http qem  
u- qem s://el u-  
and- u- astis and-  
chef- and- ys.c chef-  
insp chef- om/ insp  
ec/& insp buil ec/&  
title ec/& ding title  
=Bu t=B - =Bu  
ildin uildi and- ildin  
g ng testi g  
and and ng- and  
testi testi base testi  
ng ng - ng  
base base imag base  
imag imag es- imag  
es es for- es  
for for kube for  
Kub Kub rnet Kub  
erne erne es- erne  
tes tes clust tes  
clust clust er- clust  
er er node er  
node node s- node  
s s with s

with with - with  
 Pack Pack pack Pack  
 er, er, er- er,  
 qem qem qem qem  
 u u u- u  
 and and and- and  
 Chef Chef chef- Chef  
 InSp InSp insp InSp  
 ec) ec) ec/) ec)

This post will detail an approach for building a base image on top of Ubuntu to be used as a start image for provisioning Kubernetes nodes (masters or workers) running as virtual machines. We will be using packer to automate the build process and to ensure basic compliance testing with InSpec tests. The target audience are people who are/will be running non-managed Kubernetes clusters or people that are looking into how to leverage Packer as a tool for automated, reproducible and tested virtual machine images.

The blog post will cover:

- a brief introduction to Packer concepts
- a walk-through of the steps needed to build a base image for a Kubernetes node
- how to perform basic testing with Chef InSpec

The source files are available at: <https://github.com/elastisys/ck8s-base-vm> (<https://github.com/elastisys/ck8s-base-vm>)

## A brief introduction to Packer concepts

Packer.io is an open source utility which helps the automating the build and deployment of machine images, targeting multiple virtualization platforms such as public cloud providers or container environments. An image may contain pre-baked operating systems and various software packages pre-configured. A Packer build template is a JSON object describing a build pipeline to be executed by Packer.

It typically contains:

- a user variables section;
- builders – components that produce an image targeting a specific platform (AWS, VMWare, qemu etc.);
- provisioners – components that install and configure software, environment values, users etc. by relying on various tools (bash scripts, Ansible, Chef, Puppet etc.). Provisioners may be executed in the context of one or more builders.
- communicators: each builder configures a communicator (ssh or winrm), used to perform the tasks specified by a provisioner or a post-processor. By default ssh is used to establish a connection to the virtual machine spawned by a builder. Packer is using this connection to upload files, execute bash scripts etc.
- post-processors – components that are executed after a builder or another post-processor that act on the created artifacts or produce new ones. Post-processors can compute SHA256 checksums or import an OVA image as an AMI into Amazon EC2

## Building a base image for Kubernetes nodes using Packer

The initial infrastructure deployment of a Kubernetes cluster is a perfect use-case for Packer, as it can assist with the build of a Linux image that is optimized for the efficient provisioning of several virtual machines that will act as Kubernetes master or worker nodes. Following the official [Kubernetes documentation \(https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/#before-you-begin\)](https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/#before-you-begin)

- a system running a Linux kernel with at least 2 GB of RAM and minimum 2 CPUs
- each instance deployed from our image must have unique hostname, MAC address, machine-id/product\_uuid
- a [short list \(https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/#check-required-ports\)](https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/#check-required-ports) of inbound TCP ports should be accessible
- swap disabled
- newer Linux distributions that ship `iptables` with `nftables` back-end should have `iptables` running in legacy mode.

We will write a Packer template that will start from an Ubuntu 18.04 LTS image, spin off a virtual machine, install and configure it according to our requirements, export it as .qcow2 disk image, compute checksums and run a series of basic tests that will verify that a virtual machine provisioned from the exported disk images does contains all of the required binaries and will probe for the required ports to be accessible over TCP.

Although Packer is providing various builders that exposes user interfaces, such as VMWare or VirtualBox, our build will use qemu/kvm for the simplicity of deployment and to gain advantage of the performance a low overhead, headless build offers.

In order to be able to run the build plan the following dependencies are needed:

- An Ubuntu Bionic 18.04 LTS box
- Packer 1.5+ <https://www.packer.io/downloads.html> (<https://www.packer.io/downloads.html>)
- KVM <https://help.ubuntu.com/community/KVM/Installation> (<https://help.ubuntu.com/community/KVM/Installation>)
- Chef InSpec <https://www.inspec.io/downloads/> (<https://www.inspec.io/downloads/>)

Once the dependencies are installed, the first step is to add a builder block in the `builders` section of our `baseos.json` build template:

```
{
  "description": "Base OS VM - Ubuntu 18.04 LTS",
  "variables": {
    "disk_size": "20480",
    "iso_url": "http://cloud-images.ubuntu.com/releases/bionic/release/ubuntu-18.04-server-cloudimg-amd64.img",
    "iso_checksum": "a720c34066dce5521134c0efa63d524c53f40c68db24cf161d759356a24aad0e",
    "vm_name": "baseos.qcow2",
    "ssh_username": "ubuntu"
  },
  "builders": [
```

```

{
  "name": "baseos",
  "type": "qemu",
  "accelerator": "kvm",
  "vm_name": "{{ user `vm_name` }}",
  "format": "qcow2",
  "iso_url": "{{ user `iso_url` }}",
  "iso_checksum": "{{ user `iso_checksum` }}",
  "iso_checksum_type": "sha256",
  "disk_image": true,
  "disk_size": "{{ user `disk_size` }}",
  "disk_interface": "virtio-scsi",
  "disk_discard": "unmap",
  "disk_compression": true,
  "skip_compaction": false,
  "headless": true,
  "ssh_username": "{{ user `ssh_username` }}",
  "ssh_private_key_file": "~/.ssh/id_rsa",
  "ssh_port": 22,
  "ssh_wait_timeout": "10000s",
  "shutdown_command": "echo 'shutdown -P now' > /tmp/shutdown.sh; sudo -S sh '/tmp/shutdown.sh'",
  "http_directory": "cloud-init/baseos",
  "http_port_min": 9000,
  "http_port_max": 9100,
  "vnc_bind_address": "0.0.0.0",
  "vnc_port_min": 5900,
  "vnc_port_max": 5900,
  "qemuargs": [
    [ "-m", "2048M" ],
    [ "-smp", "2" ],
    [ "-smbios", "type=1,serial=ds=nocloud-net;instance-id=packer;seedfrom=http://{{ .HTTPIP }}:{{ .HTTPPort }}/\" ] ] }
  ],
},
"provisioners": [],
"post-processors": []

```

In short, the builder configuration instructs packer to use an Ubuntu 18.04 cloud image as a base file system in order to start a virtual machine using qemu with kvm acceleration. It also provides specifications such as the desired `disk_size` for the provisioned virtual machine, the disk virtual interface to be used (`virtio-scsi`), whether the resulting virtual disk should be compressed and compacted when the builder is finished, as well as ssh authentication details for the ssh communicator.

There are a couple of implementation details to notice. First, the builder does not start from a released ISO image of Ubuntu, but instead it uses a cloud image as a base. Release ISO distributions would need a customized boot command and would make use of pre-seed files to orchestrate the installation. This would require more implementation work while the build would last significantly longer (for example the builder will have to replace the preset boot command at a speed of one character per second). Cloud-ready images are base file systems optimized to be used for provisioning cloud instances and have less packages deployed to ensure smaller image sizes. Ubuntu cloud images such as the server release or the minimal release are shipped with the pre-installed packages required by most cloud platforms. These include `cloud-init` (used to dynamically push configurations to an instance at first boot) and `cloud-guest-utils` (containing `growpart` tool needed

for extending partitions). If you want to find out more about building from ISO images, there are detailed steps presented in the official [Packer documentation \(https://www.packer.io/guides/automatic-operating-system-installs/preseed\\_ubuntu.html\)](https://www.packer.io/guides/automatic-operating-system-installs/preseed_ubuntu.html)

Secondly, the builder is specifying a couple of arguments for the qemu binary. These include options that specify resource allocation (CPU cores and RAM) and a `-smbios` option that sets custom data into the SMBIOS table by using its `serial` field: `"-smbios", "type=1,serial=ds=nocloud-net;instance-id=packer;seedfrom=http://{{ .HTTPIP }}:{{ .HTTPPort }}/"`. Our starting image is running cloud-init, thus cloud-init data sources can be used at first boot to configure the credentials needed by packer's ssh communicator (username, ssh public key). SMBIOS is used to configure [NoCloud](https://cloudinit.readthedocs.io/en/latest/topics/datasources/nocloud.html)

[\(https://cloudinit.readthedocs.io/en/latest/topics/datasources/nocloud.html\)](https://cloudinit.readthedocs.io/en/latest/topics/datasources/nocloud.html) data source of cloud-init and define the endpoint where the cloud-init configuration will be accessible. When qemu boots the image, cloud-init will attempt to fetch cloud configuration data from an endpoint provided by a server created by Packer during the build process. This http server will serve the path defined by `http_directory`, which is set to a local folder containing cloud-init files, over a local address and port that are accessible in the builder configuration as template variables: `{{ .HTTPIP }}` and `{{ .HTTPPort }}`.

Next step is to actually create the local folder containing the cloud-init configurations, as configured in the `http_directory` property of the qemu builder. We will create a nested folder structure `./cloud-init/baseos` at the same level as our Packer template file. The NoCloud cloud-init data source expects to find two files `meta-data` and `user-data`. The `meta-data` file should be used to represent any information commonly found in an [Amazon EC2 metadata \(https://cloudinit.readthedocs.io/en/latest/topics/datasources/ec2.html\)](https://cloudinit.readthedocs.io/en/latest/topics/datasources/ec2.html) service, while the `user-data` file is simply a user-data YAML file containing [cloud-config syntax. \(https://cloudinit.readthedocs.io/en/latest/topics/examples.html\)](https://cloudinit.readthedocs.io/en/latest/topics/examples.html)

There is no need to push meta-data to the virtual machine, so create an empty `meta-data` file. If you do not have an SSH key pair, it is time to generate one. Next add the following cloud configuration to the `user-data` file and paste your public key under `ssh-authorized-keys`:

```
#cloud-config
users:
name: ubuntu
sudo: ALL=(ALL) NOPASSWD:ALL
ssh-authorized-keys:
ssh-rsa ...
apt:
preserve_sources_list: true
package_update: false
```

This simple cloud configuration is adding password-less `sudo` privileges to the default user `ubuntu`, it injects a public key to be used for SSH authentication, freezes the sources for package updates and disables the package updates in order to ensure a minimal reproducibility of the image build process.

Having the builder configured, the following step is to write the provisioners that will customize the images. We will follow the official Kubernetes documentation and install:

- install and configure a container runtime: Docker

- the Kubernetes administration packages: `kubeadm`, `kubelet`, `kubectl`

Additionally we clean-up the image and reset the state of cloud-init in order to ensure that future virtual machine provisioned from our image will execute cloud configurations correctly and will satisfy the requirements set by Kubernetes for unique machine-ids and MAC addresses. There are four provisioners to be executed in the context of provisioner `baseos`:

```
"provisioners": [
{
  "type": "shell",
  "script": "./scripts/base-setup.sh",
  "expect_disconnect": true,
  "environment_vars": [
    "DEBIAN_FRONTEND=noninteractive"
  ],
  "execute_command": "sudo -S bash -c '{{ .Vars }} {{ .Path }}'",
  "only": ["baseos"] },
{
  "type": "shell",
  "script": "./scripts/install-docker.sh",
  "environment_vars": [
    "DEBIAN_FRONTEND=noninteractive"
  ],
  "execute_command": "sudo -S bash -c '{{ .Vars }} {{ .Path }}'",
  "only": ["baseos"] },
{
  "type": "shell",
  "script": "./scripts/install-k8s-tools.sh",
  "environment_vars": [
    "DEBIAN_FRONTEND=noninteractive"
  ],
  "expect_disconnect": true,
  "execute_command": "sudo -S bash -c '{{ .Vars }} {{ .Path }}'",
  "only": ["baseos"] },
{
  "type": "shell",
  "script": "./scripts/clean-up.sh",
  "environment_vars": [
    "DEBIAN_FRONTEND=noninteractive"
  ],
  "expect_disconnect": true,
  "execute_command": "sudo -S bash -c '{{ .Vars }} {{ .Path }}'",
  "only": ["baseos"] },
]
```

All our provisioners are of `type: "shell"`, will execute only for the `baseos` builder and will run file-based shell scripts from the path described by their `scripts` option, using `bash` as specified in the `execute_command` property. The command itself uses two template variables `{{ .Vars }}` and `{{ .Path }}`. It is possible to define environment variables using the `environment_vars` option, such as `DEBIAN_FRONTEND=noninteractive` which will enable a non-interactive front-end for package installations, and these will be made available in the context of the command to be executed as the `{{ .Vars }}` template variable. `{{ .Path }}` will reference the path to the script to be executed.

Some of the scripts might trigger system reboots which would normally make packer exit with a build failure, unless the provisioner is configured with `expect_disconnect: true` options.

The provisioner scripts go through all the stages required to build our

The provisioner scripts go through all the stages required to build an image:

- `base-setup.sh`: disables unattended upgrades to avoid unpredictable future behavior of Kubernetes nodes due to package updates. Given that both `machine-id` and `cloud-init` will be initialized and will **trigger a systemd “factory reset (<https://www.get-edi.io/11-Traps-to-Avoid-When-Building-Debian-Images/>)”** at next boot, the disabled services are also added to `/lib/systemd/system-preset`. The script also turns swap off, removing swap partitions and triggering a reboot to allow the clean up of all files and locks imposed by the system swap.
- `install-docker.sh`: follows the official Kubernetes guide to install Docker container runtime and its dependencies and configures Docker to use `systemd` as cgroup driver.
- `install-k8s-tools`: follows the official Kubernetes guide to install the cluster administration packages (`kubeadm`, `kubelet`, `kubectrl`). At the time of writing there is no Kubernetes release for Ubuntu Bionic, thus Ubuntu Xenial releases are used.
- `clean-up.sh`: cleans up `machine-id`, resets the run state of `cloud-init` to allow for new cloud configurations to be pushed whenever a virtual machine is provisioned from the image.

As a final step, we finalize the template in `baseos.json` by adding a post-provisioner that will calculate the md5 and sha256 checksums of the resulting `.qcow2` image and will save them into a file in the same location as the built image:

```
"post-processors": [  
  {  
    "type": "checksum",  
    "checksum_types": [  
      "sha256",  
      "md5"  
    ],  
    "output": "./output-baseos/{{ user `vm_name` }}_{{.ChecksumType}}.checksums",  
    "only": ["baseos"] }  
]
```

The packer build template can be run using a simple shell command, but in order to make the workflow smoother we will create a Makefile in the same location as the `baseos.json` file and define run and clean-up targets:

```
baseos: baseos.json  
PACKER_LOG=1 CHECKPOINT_DISABLE=1 packer build -only=baseos baseos.json > baseos-build.log; \  
case "$$" in \  
  0) \  
    echo "Base image created." \  
    ;; \  
  1) \  
    echo "Image already present. Run make clean to remove all artifacts." \  
    ;; \  
  *) \  
    echo "Unhandled error" \  
    ;; \  
esac;  
  
clean:  
rm -rf output-baseos*  
rm baseos-build*.log
```



Running the build template using `make baseos` should save the resulting `.qcow2` image and its checksums in `./output-baseos`, as well as log the whole process in a file named `baseos-build.log`.

## Compliance testing of Packer images with Chef InSpec

Having a `.qcow2` image successfully built does not guarantee its functional state, especially when it is designed to be run in complex cluster deployments such as Kubernetes. We will ensure a minimal compliance testing stage using Chef InSpec to design a test suite that will use a virtual machine provisioned using the `baseos.qcow2` image in order to:

- check for the existence of the required binaries such as `docker`, `kubeadm` etc.
- verify that all required ports are accessible once `kubeadm` has been initialized, thus the node is ready to form a Kubernetes cluster.

This test plan is just a starter kit and production deployments should ensure full conformance and compliance testings according to community best practices.

We will add a new builder to our Packer template, that will use `baseos.qcow2` as a base image to provision a test virtual machine:

```
{
  "name": "baseos-test",
  "type": "qemu",
  "accelerator": "kvm",
  "vm_name": "test-baseos.qcow2",
  "headless": true,
  "output_directory": "./output-baseos-test",
  "disk_image": true,
  "use_backing_file": true,
  "shutdown_command": "echo 'shutdown -P now' > /tmp/shutdown.sh; sudo -S sh '/tmp/shutdown.sh'",
  "iso_url": "output-baseos/baseos.qcow2",
  "iso_checksum_type": "sha256",
  "iso_checksum_url": "./output-baseos/{{ user `vm_name` }}_sha256.checksums",
  "disk_compression": false,
  "skip_compaction": true,
  "ssh_username": "{{ user `ssh_username` }}",
  "ssh_private_key_file": "~/ssh/id_rsa",
  "ssh_port": 22,
  "ssh_wait_timeout": "10000s",
  "vnc_bind_address": "0.0.0.0",
  "vnc_port_min": 5900,
  "vnc_port_max": 5900,
  "http_directory": "cloud-init/baseos-test",
  "qemuargs": [
    [ "-m", "2048M" ],
    [ "-smp", "2" ],
    [ "-smbios", "type=1,serial=ds=nocloud-net;instance-id=packer;seedfrom=http://{{ .HTTPIP }}:{{ .HTTPPort }}" ] ] ] }
```

We want to avoid for changes to be written to our `baseos.qcow2` file, so we make use of the backing file featured in `qemu` and configure the builder with `"use_backing_file": true` option. As a result, `qemu` will start a virtual machine using two disk images: the first one will be a clone of our previously created `baseos.qcow2` and will work as a backing file in read-only mode. Any modified

blocks originating on the backing file will be written into a second disk image, which will act as a writable layer on top of the read-only backing file. We can thus reconfigure the test image and run additional provisioners that will mutate the configuration as needed without altering the disk image that we are testing.

The `baseos-test` builder reads cloud-init configuration files from the directory `./cloud-init/baseos-test` in order to inject a SSH public key associated with the `ubuntu` user. Once the virtual machine has finished bootstrapping, a first provisioner will run a `init-test-node.sh` script, which will execute `kubeadm init` to initialize the VM as a Kubernetes node, before the actual tests will be run by an InSpec provisioner:

```
{
  "type": "shell",
  "script": "./scripts/init-test-node.sh",
  "environment_vars": [
    "DEBIAN_FRONTEND=noninteractive"
  ],
  "execute_command": "sudo -S bash -c '{{ .Vars }} {{ .Path }}'",
  "only": ["baseos-test"] },
{
  "type": "inspec",
  "pause_before": "30s",
  "timeout": "10m",
  "profile": "./tests/baseos-profile",
  "inspec_env_vars": [ "CHEF_LICENSE=accept" ],
  "extra_arguments": [ "-l", "debug" ],
  "only": ["baseos-test"] }
```

InSpec is an open-source framework for testing and auditing applications and infrastructure, by comparing desired system states defined in InSpec profiles with the actual state of a system. Our `inspec` provisioner will run the tests described in `./tests/baseos-profile`. InSpec uses a Ruby based Domain Specific Language to nominate specific system resources such as files, services, ports and specify conformity checks for each of these:

```
describe file(arguments...) do
  ...
end
```

Our tests are defined in `./tests/baseos-profile/controls/default.rb` and follow the official InSpec code conventions. A first set of tests check for specific binaries to exist, by describing them as commands that are expected to be callable and exit with status 0.

```
# Check binaries
describe command('which docker') do
  its(:exit_status) { should eq 0 }
  its(:stdout) { should match('/usr/bin/docker') }
end

ck8s_binaries = ['kubectll', 'kubeadm', 'kubelet'] ck8s_binaries.each do |ck8s_binary|
  describe command(ck8s_binary).exist? do
    it { should eq true }
  end
end
```

The second series of tests will define a set of ports that are used by Kubernetes control-plane and check if they are accessible over TCP:

```
# Verify control-plane ports. See: https://kubernetes.io/docs/setup/production-
environment/tools/kubeadm/
tcp_ports = [2379, 2380, 6443, 10250, 10251, 10252] tcp_ports.each do |tcp_port|
  describe port(tcp_port) do
    it {should be_listening}
    its('protocols') {should cmp 'tcp'}
  end
end
```

The final step is to add a new target to our Makefile in order to run the tests using `make test`:

```
test: baseos
PACKER_LOG=1 CHECKPOINT_DISABLE=1 packer build -only=baseos-test baseos.json > baseos-build-
tests.log
```

This concludes the walk-through on how to build and test a base image for Kubernetes nodes. Check the full source of the Packer build template [here](https://github.com/elastisys/ck8s-base-vm) (<https://github.com/elastisys/ck8s-base-vm>) and stay tuned on our website for more technical blog posts about working with Kubernetes.



(http   
s://w (http   
ww.l s://n (http (http  
inke ews. s://t s://r  
din.c yco witt eddi  
om/c mbi er.c t.co  
ws/s nato om/i m/su  
hare r.co nten bmit  
? m/su t/tw ?  
url= bmit eet? url=  
http link? text http  
s://el u=h =Bu s://el  
astis ttps: ildin astis  
ys.c //ela g ys.c  
om/ stisy and om/  
buil s.co testi buil  
ding m/b ng ding  
- uildi base -  
and- ng- imag and-  
testi and- es testi  
ng- testi for ng-  
base ng- Kub base  
- base erne -  
imag - tes imag

es- imag clust es-  
for- es- er for-  
kube for- node kube  
rnet kube s rnet  
es- rnet with es-  
clust es- Pack clust  
er- clust er, er-  
node er- qem node  
s- node u s-  
with s- and with  
- with Chef -  
pack - InSp pack  
er- pack ec at er-  
qem er- http qem  
u- qem s://el u-  
and- u- astis and-  
chef- and- ys.c chef-  
insp chef- om/ insp  
ec/& insp buil ec/&  
title ec/& ding title  
=Bu t=B - =Bu  
ildin uildi and- ildin  
g ng testi g  
and and ng- and  
testi testi base testi  
ng ng - ng  
base base imag base  
imag imag es- imag  
es es for- es  
for for kube for  
Kub Kub rnet Kub  
erne erne es- erne  
tes tes clust tes  
clust clust er- clust  
er er node er  
node node s- node  
s s with s  
with with - with  
Pack Pack pack Pack  
er er er- er



## Services

[Managed Services\(/\)](#)

[Consulting Services\(/consulting/\)](#)

[On-Prem Compliant Kubernetes\(https://elastisys.com/kubernetes-on-premise/\)](https://elastisys.com/kubernetes-on-premise/)

[Training\(/training/\)](#)

## Resources

[Tech Blog\(https://elastisys.com/blog/\)](https://elastisys.com/blog/)

[Documentation\(https://elastisys.io/compliantkubernetes/\)](https://elastisys.io/compliantkubernetes/)

[Legal Documents\(/legal/\)](#)

[Terms of Service\(https://elastisys.com/terms-of-service/\)](https://elastisys.com/terms-of-service/)



([https://elastisys.com/wp-content/uploads/2022/02/certifikat\\_27001\\_sv-eng.pdf?x40195](https://elastisys.com/wp-content/uploads/2022/02/certifikat_27001_sv-eng.pdf?x40195))



WITH SUPPORT FROM

[www.datacenterinnovationregion.se](https://www.datacenterinnovationregion.se/) (<https://www.datacenterinnovationregion.se/>)