# MING: Model- and View-Based Deployment and Adaptation of Cloud Datacenters

Ta'id Holmes(✉)

Infrastructure Cloud, Deutsche Telekom Technik GmbH, Darmstadt, Germany
t.holmes@telekom.de

**Abstract.** For the configuration of a datacenter from bare metal up to the level of infrastructure as a service (IaaS) solutions, currently, there is neither a standard nor a common datamodel that is understood across deployment automation tools. Following a model- and view-based approach, MING (明) aims at holistically describing cloud datacenters. Establishing a respective metamodel, it supports different stakeholders with tailored views and permits utilization of arbitrary deployment tools for providing the basic cloud service model. In addition to initial deployments, it targets (model-based) adaptation of datacenters for covering operational use cases such as extending a cloud with additional resources and for providing for software upgrades and patches of the deployed solutions.

**Keywords:** Adaptation · Cloud · Code generation · Configuration · Datacenter · Deployment · DSL · IaaS · MBE · Metamodel · OpenStack · SDN

## 1 Introduction

The cloud computing paradigm continues to change the way end-users consume services and communicate (cf. [31]). At the same time service providers adapt (*cloudify*) software services (cf. [2]) for profiting from the benefits of cloud computing (cf. [18]). Among them are the possibility to meet changing workloads through elastic scaling, to make efficient use of resources, e.g., through multitenancy, an on-demand usage and pricing model, as well as defined models and roles in regard to infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS) (cf. [33]).

Thus, cloud computing penetrates all levels from datacenters (DCs) to end-users. On an infrastructure level cloud computing offers (virtualized) hardware resources and network capabilities as IaaS. This impacts the design and deployment of DCs.

For this reason the planning, implementation, and operation of DCS has changed. Above all, cloud DCs distinguish themselves from traditional DCs by

This is an extended version of a former contribution [23] incorporating post-deployment aspects.


M. Helfert et al. (Eds.): CLOSER 2016, CCIS 740, pp. 317–338, 2017.
DOI: 10.1007/978-3-319-62594-2_16

having an IaaS solution deployed. The IaaS solution establishes the correspondent service model while managing the DC resources. These resources comprise computational power from central processing units (CPUs), random-access memory (RAM), storage in the form of objects stores or block devices from hard drive disks (HDDs) and solid-state drives (SSDs), and network interfaces. All of these need to be registered and managed by the IaaS solution.

OpenStack [42] is a popular IaaS solution with a big community and support across industries. For easing the installation and deployment of OpenStack various automated deployment technologies and tools exist. Often they are provided from an operating system (OS) vendor as a kind of value proposition.

Software-defined networking (SDN) (cf. [32]) and storage (cf. [56]) solutions may be deployed in addition to OpenStack. Providing a scalable networking architecture beyond other benefits, the former eases the planning of DCs. The later offers storage as a service (e.g., in form of a block device or object storage) towards OpenStack.

For realizing the installation of bare machines and the deployment of SDN, storage, and IaaS solutions, currently, various information needs to be aggregated in configurations by experts who are familiar with the technologies. Often the information relates to different aspects and is scattered and tangled and needs to be kept consistent. Changes in the design or characteristics of a DC may impact the configuration fundamentally: e.g., number of availability zones in a DC, dedication of a certain node to some aggregate[1], or networking.

Ideally, it would be possible to describe the various aspects of a cloud DC (e.g., the hardware and the networking) and its deployment (i.e., OSs and the services) conceptually in a domain-specific language (DSL) that is tailored towards the respective experts so that from such information as contained in the respective views the automated DC deployment can take place. While several deployment tools exist (that in fact can all be made use of following the model-based approach) there is no technology agnostic datamodel for specifying a DC deployment that is understood by tools.

Therefore, MING (明), a model-based approach, is proposed: it permits the model- and view-based description of DCs and respective IaaS deployments by means of a platform-independent model (PIM). This way, separation of concerns (SoC) is realized supporting different stakeholders. Also, integration with existing tools is realized using code generation. Finally, for supporting operational aspects MING aims at supporting post-deployment adaptation scenarios.

The remainder is structured as follows: The following section further introduces the context by explaining tasks when deploying and maintaining a cloud DC. Prior to presenting the MING approach in Sect. 5, Sect. 3 relates to the state of the art and Sect. 4 presents some background on model-based engineering (MBE).

[1] In order to profit from Enhanced Platform Awareness (EPA) host aggregates can be defined in OpenStack. Resources can be allocated within such aggregates for hosting services that want to make use of features such as hugepages, Non-Uniform Memory Access (NUMA), or CPU pinning for achieving high throughput.

Next, Sect. 6 presents some details of the current prototype. Section 7 discusses on the benefits, risks, and limitations and Sect. 8 concludes.

## 2 Automated Datacenter Deployment

The planning, setup, implementation, and operation of a DC comprises multiple activities involving hard- and software. Prior to focusing on the latter, i.e., the automated software installation and deployment, this section first looks at the structure of DCs.

### 2.1 Structure of Datacenters

Nowadays, a DC follows a Leaf-Spine topology (cf. [1]) and is connected to the core network through datacenter routers (DCRs). For this, each top of the rack router (ToR) is fully meshed to the spine layer. Establishing predictability in latency as well as redundancy for achieving high availability, any spine router is thus connected to every DCR as well as ToR. Similarly, a dedicated operations, administration, and management (OAM) network comprises bottom of the rack routers (BoRs).

Figure 1 depicts a simple metamodel for DCs (that is also part of the Ming metamodel, cf. Fig. 2). A DataCenter comprises one or several AvailabilityZones. These may be fire compartments that are separated from each other. Each AvailabilityZone contains Racks for mounting equipment such as routers and servers (Node). A server comprises network interface controllers (NICs), CPU, RAM, and storage in form of HDDs and/or SSDs. Each NIC has a unique media access control (MAC) address. For networking (cf. Layer 3 of the Open Systems Interconnection (OSI) reference model [58]) a NIC will be configured at some stage with an Internet Protocol [10] (IP) address, e.g., using Dynamic Host Configuration Protocol [17] (DHCP). Within a Network (i.e., ip and netmask) a NIC has a particular DeviceID (e.g., the last bit(s) of the address).

Installation nodes are used for bootstrapping the DC deployment. Generally it is possible to group servers into different categories: Storage nodes contain a large amount of storage capacity while Compute nodes have high computational power. Network nodes comprise fiber optical NICs for high bit rates. Finally, Management nodes are dedicated for hosting IaaS services (see also Sect. 2.2).

Intelligent Platform Management Interface [25] (IPMI) may provide an administrative access to the servers through a dedicated network. Besides, all servers may be part of multiple physical or virtual (vlanID) networks. For example, storage nodes may have a backend network for replication in addition to a frontend network for the data.

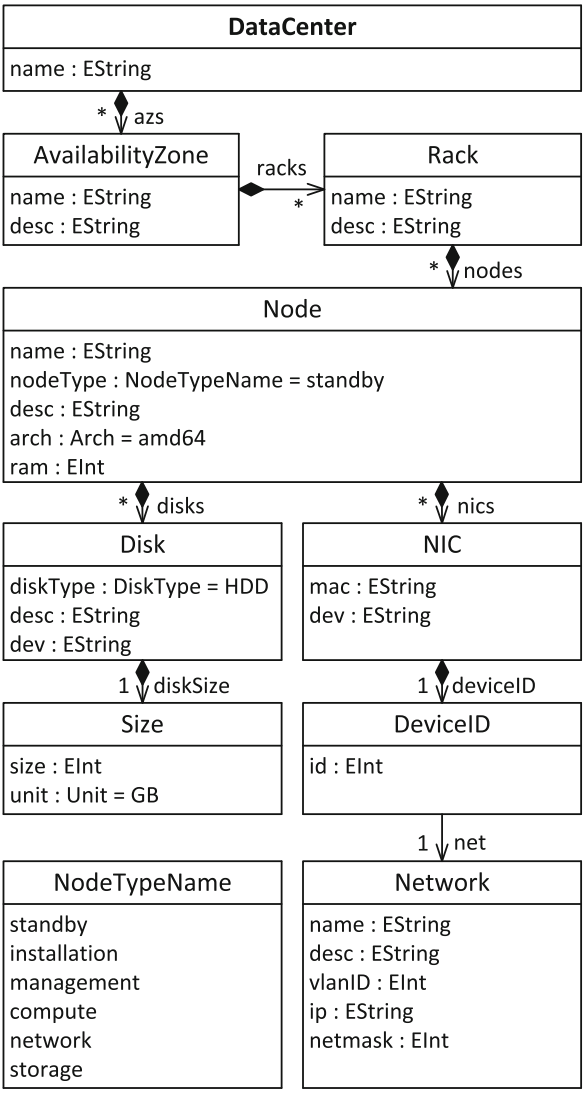


**Fig. 1.** A datacenter metamodel.

### 2.2 Software Installation

Given a DC with a completed physical setup including networking and cabling, deployment of an SDN solution, and configuration of the network devices installation of the bare machines can take place (see also Sect. 3.1). That is, on each node a base OS is installed over the net, e.g., using IPMI and Preboot Execution Environment [24] (PXE) together with a DHCP server. Yet, some information needs to be collected beforehand and placed at the DHCP server such as the

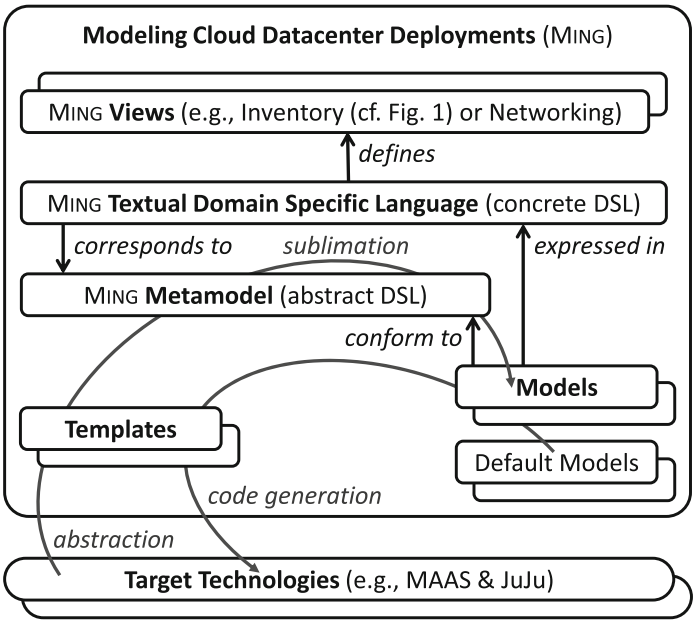


**Fig. 2.** Approach overview: the MING framework.

MAC addresses of the NICs. This information may be discovered automatically. For the software installation a local mirror can be made use of (see also Sect. 2.3).

Next, IaaS services can be installed (see also Sect. 3.2). STORAGE servers may deploy Ceph [56], a distributed object store. MANAGEMENT nodes host services such as OpenStack Identity Service (Keystone), OpenStack Compute (Nova) cloud controller, OpenStack Orchestration (Heat), and OpenStack Dashboard (Horizon). COMPUTE nodes run Nova. Finally, OpenStack Networking (Neutron) is used for NETWORK nodes and integrates with a deployed SDN solution through a respective plugin.

### 2.3 Software Projects, Artifacts, and Version Control

For such a described deployment, several software projects and subprojects have to be considered. In this context, we need to distinguish between up- and downstream projects. Upstream projects hold their own repositories where (community) development takes place. Downstream projects take releases delivered by upstream projects and customize these, e.g., for a respective OS or a container engine. Thus, they produce software artifacts such as Debian or RedHat Package Manager (RPM) packages or Docker [16], [4] images.

For an IaaS provider it is crucial to know what versions of the respective projects are (to be) deployed and to control the software update and upgrade. Addressing this requirement, a local mirror may be deployed on installation

nodes that stores all the software artifacts. Such a mirror also permits the provider to gain independence towards external services and resources. For this, prior to an automated installation for a production environment, the provider holds a self-contained set of data (software images, packages, configuration management (CM) artifacts). Acting as a caching proxy in a test environment, such a mirror may be populated in the course of an initial installation. This way, all the required software packages can be captured and identified easily. Versions that shall be foreseen for production can be frozen in a review, e.g., using a tagging mechanism. This is important in regard to reproducibility as required for production.

Once a new version of a software artifact becomes available, e.g., in public mirrors, an operations team is notified and the artifact is scheduled for testing and in case of success promoted to the next delivery stage.

### 2.4 Datacenter Scaling

One of the distinguishing features of cloud applications is their ability to scale horizontally (cf. [18]). That is, resources are provisioned dynamically on-demand (referred to as *elastic scaling*) when experiencing changing workloads. It is only consequent to translate this principle into the context of Dcs and apply it to physical resources as well.

One of the implications is that an IaaS deployment may start small and grow bigger over time. As a result, a DC is only partially deployed with racks, initially. The more and more applications are onboarded over time additional racks are ordered and integrated into the DC and the IaaS deployment. This iterative deployment of DCs is interesting from an economic point of view. As mentioned, a Leaf-Spine topology and SDN eases the scaling within a DC. In the context of OpenStack the Tuskar project of TripleO offers an approach how to address the scaling of IaaS deployments (see also Sect. 3.2).

Generally, new resources such as compute or storage nodes shall be integrated into the cloud, e.g., OpenStack or Ceph, posterior to an initial deployment. Also it shall be possible to seamlessly replace faulty hardware.

Having briefly introduced the deployment of cloud DCs, the following section discusses different existing deployment tools for realizing the installation.

## 3 State of the Art

Prior to presenting the Ming approach let us first have a look at the state of the art. At the end of this section, a current shortcoming of the existing deployment technologies is summarized for positioning the contribution of Ming. As outlined in the previous section there are at least two distinct phases in the deployment of DCs: bare machine and IaaS installation with some of the tools focusing on the former and others on the latter.

### 3.1 Bare Machine Installation

The following selection of tools and projects specializes on installing a base OS on bare machines (cf. [11] for the evaluation of some frameworks).

**Cobbler** [13] is a lightweight build and provisioning system for the deployment of physical and virtual machines. Objects and variables are used for configuring the provisioning. These are then applied in templates, e.g., for generating preseed files. This way, i.e., through templates, Cobbler also integrates with Kickstart [44]. Generally, integration with existing tools and CM systems is encouraged. In addition to a command-line interface (CLI) there is a also a web user interface (UI). Cobbler is currently used by Compass and Fuel (see Sect. 3.2).

**Fully Automatic Installation** [19,28,29] – with a particular focus on unattended automated installations – builds, as Cobbler, on top of technologies such as PXE, DHCP, and Trivial File Transfer Protocol [48] (TFTP). Originally focusing on Debian-based [47] distributions, FAI has been adopted for CentOS [50]. It realizes profiles in addition to a class concept that can help to describe complex setups.

**OpenStack Bare Metal Provisioning** [39] **(Ironic)** is used for the provisioning of physical machines within OpenStack. Thus, in contrast to the other tools of this category, it is not a self-contained system. Its functionality is used by TripleO and will also be relied on by Fuel.

**Metal as a service** [8] **(MAAS)** is used for the provisioning of OSs in combination with JuJu [7] and Charms (see Sect. 3.2). Similar to Cobbler and FAI, a MAAS server acts as a DHCP server for the provisioning of machines. Hosts (physical or virtual) can be put under control of a MAAS server. Configuration such as MAC to IP mapping can be done in a YAML Ain't Markup Language (YAML) file (see also Sect. 6).

### 3.2 OpenStack Installation

The automated provisioning, configuration, and installation of services is addressed by CM systems such as Ansible [14], Chef [12], JuJu, or Puppet [43]. Thus, after each node has been installed with an OS, the installation and configuration of IaaS services can be realized using CM. For the deployment of OpenStack there is a variety of existing tools:

**Compass** [37] supports different CM systems through a plugin architecture. By establishing abstraction layers, it also decouples resource discovery and bare metal installation. Besides, it facilitates operations support system (OSS) integration through a northbound interface (NBI).

**Crowbar** [15] is a project that relies on the Chef CM system for the deployment of applications such as OpenStack or Hadoop [49]. In contrast to other solutions of this category it does not presume but also realizes bare metal installation and comes with a web UI.

**Fuel** [38] offers a web UI frontend for the deployment of OpenStack in addition to a CLI. Cobbler is currently used under the hood, yet, migration to OpenStack Bare Metal Provisioning (Ironic) is intended. Puppet is used for CM. Some features comprise the automated discovery of nodes and the possibility to perform pre-deployment checks.

**JuJu** is an orchestration technology that is also used for MAAS. JuJu bundles describe the orchestration of applications. Charms, classified by Wettinger et al. [57] as environment-centric artifacts, deploy the actual OpenStack services of JuJu bundles. Thus, the deployment of OpenStack is specified in a JuJu YAML file referencing different charms that are related to respective upstream projects.

**Packstack** [41] provides Puppet modules for OpenStack projects. Using Puppet for CM, the various OpenStack services can be deployed. Thus, some frontend deployment tools such as RPM Distribution of OpenStack [45] (RDO) (see below) make use of Packstack. Currently, distributions based on RPM are supported.

**RDO** is a web-based deployment tool based on Foreman [27], a Ruby on Rails [21] application and frontend for the CM with Puppet. Therefore Packstack is used.

**TripleO** [40] is an exception to the CM-based solutions. Instead of relying on such, TripleO aims at realizing the functionality using OpenStack's own cloud features for facilitating installation, management, and operation. For this, a deployment cloud (a.k.a. undercloud) needs to be setup first. Using Ironic workload cloud(s) (a.k.a. overcloud(s)) are deployed. The deployment and configuration of nodes is realized using Heat. For this golden images need to be prepared. These consist of a base OS with elements on top (resembles FAI class and profile concept). During provisioning a node will configure itself using the parameters from a Heat Orchestration Template (HOT) that constitutes the deployment plan. Finally, an overcloud can be scaled using the Tuskar subproject.

### 3.3 Positioning and Contribution of Ming

Currently, there is neither a standard nor a common datamodel for the configuration of an OpenStack deployment in a cloud DC. As a result, none of the

projects exposes its configuration in a form that can be used by other projects. This however would be interesting in order to evaluate different frameworks and avoid tool dependencies. TripleO – aiming at avoiding any third party dependency for the deployment of OpenStack – is a particular case. It is using HOT for realizing the CM. This way, it decouples the configuration from the automated deployment. It may be argued that HOT is an established format that other tools could implement. Yet, this is not feasible, as it dictates orchestration through Heat. Not only Heat but also JuJu is an orchestration technology. In both cases, therefore, configuration needs to be expressed in a particular syntax and way by experts leading to the problems mentioned such as scattering and tangling.

Ming in contrast truly decouples configuration from automated deployment technologies. It declarativly permits the view-based modeling of DCs and their deployments, facilitating SoC. As a result, stakeholders that are not familiar with the used deployment technologies and/or other concerns of the deployment are supported as well. Similar in spirit with Cobbler Ming integrates with arbitrary tools and frameworks (as also envisioned by Compass) through code generation. Last but not least, it establishes a deployment tool-agnostic metamodel that serves as a datamodel for DC deployments.

## 4 Background on Model-Based Engineering

This work follows an MBE approach as outlined in the next section. Thus, this section briefly gives some background information regarding MBE. MBE is a paradigm of software engineering that establishes models as first-class entities (cf. [36]). Metamodels are used for formally describing concepts on a distinct level of abstraction (cf. [5]). Models, that need to conform to such metamodels, can be validated and are usually used in model transformations. Finally, model transformations (cf. [34]) map models such as in a model-to-model (M2M) transformation or a code generation (model-to-text (M2T) transformation). In case of forward engineering, resulting models generally conform to metamodels with a lower level of abstraction. This is for example the case when a PIM is mapped to a platform-specific model (PSM) or when code is generated.

In this work, MBE is used for establishing a metamodel that describes the IaaS deployment of a DC from bare metal. As the metamodel is agnostic towards deployment tools is can be classified as a PIM. Conforming models are transformed through code generation to target technologies.

### 4.1 Domain-Specific Languages

While metamodels are abstract in their nature, one or more (concrete) DSLs can define each a particular syntax for a metamodel (also called *abstract DSL*) that is tailored towards stakeholders of a respective domain (cf. [35,55]). This way, it is possible to express and represent models using a graphical or textual syntax as defined by the language. Models that are expressed in a DSL can

be transformed and represented in another DSL. Similarly to general purpose programming languages and compilers, models expressed in higher-level DSLs can be translated to lower-level language artifacts. In fact, models expressed in DSLs can also be mapped to general purpose programming languages.

### 4.2 View Models

The principle of SoC is proven to be a successful approach to manage complexity. In MBE, it is realized using view models. That is, different concerns are separated into distinct views. This enables different stakeholders to relate to concerns which are relevant to them more easily. Within the domain of software architecture, for example, view models have been established and standardized (cf. [26]).

In this work, a DSL that is bound to the MING metamodel defines various views that describe different aspects of the deployment such as inventory, networking, and IaaS configuration options.

## 5 Approach: Abstracting from Technologies

The approach aims at a tool agnostic, declarative specification of configuration for realizing an IaaS deployment in a cloud DC from bare machines. For this, configurations from existing provisioning and deployment technologies need to be sublimated using reverse MBE. That is, models are established through abstraction. Given a valid deployment plan DC specific values and repetitive code need to be identified in a first step.

### 5.1 Establishing Models Through Abstraction

For capturing respective information in models, a conceptual metamodel is derived and templates are created in a next step. Finally, integration with the target technologies is realized using code generation. That is, the same code is generated using the MBE approach. As a result, a conceptual modeling layer is established with sublimated configuration in form of models.

Figure 2 depicts the MING framework. As an interface for populating, expressing, and representing models, a textual DSL is bound to the MING metamodel (i.e., abstract DSL). In order to support SoC, distinct views permit the expression of different aspects. One view, e.g., covers the DC related inventory information as depicted in the metamodel shown in Fig. 1. Other views capture networking, node assignments (e.g., to an aggregate), OpenStack specific configuration, and credentials.

For supporting convention over configuration, defaults can be expressed in models too, that are applied to models in case of missing configuration. Finally, model transformation processes the (resulting) models and generates code using templates.

### 5.2 Model-Based Adaptation

The above described code generation is suitable for initial deployments. As the approach is generally limited to such, post-deployment adaptation scenarios are not supported *per se*, however. Such adaptation scenarios comprise the addition of new DC resources to the cloud, e.g., in the form of new racks or the reassignment of nodes to some aggregates as pointed out in Sect. 2.4. The latter may become necessary if free resources run low within such an aggregate, e.g., as required for services that make use of EPA features. In such a case (unused) – e.g., compute – nodes may be reassigned to a corresponding host aggregate. This implies evacuation and a complete redeployment of the node, i.e., migration of remaining resources such as server instances, a wipe-out of existing disks, a new OS installation, deployment of appropriate services, and, finally, integration into the target aggregate.

Without support of operational aspects the models – apart from describing and realizing initial deployments – would be of no further use. Yet, because of their conceptual level of abstraction and the availability of multiple views, they are also interesting for later stages of the DC lifecycle.

Ideally, it would be possible to change such models and automate a respective adaptation. In either case it would be beneficial to describe the impact for the various possible changes and simulate an adaptation. This way, cloud engineers – while changing the models – could interactively receive information on the respective impact and further guidance. Table 1 lists and describes some possible model changes based on the inventory concepts of the MING metamodel as shown in Fig. 1. Such changes can be stored in a diff-model, i.e., a model that comprises model differences between two models conforming to the same metamodel.

In case of resources it is possible to base an execution engine on processing such a diff-model (cf. [22]). That is, for every change a model transformation is performed for generating respective adaptation actions related to API calls (e.g., the provisioning of a resource).

In this regard it seems promising to adopt such an adaptation approach that permits for an incremental deployment. Yet, beyond resources the MING metamodel also comprises other concerns such as networking or configuration options. In contrast to resources that can be added or removed – just as described in each difference of a diff-model with the kind of change and the respective model

**Table 1.** Examples of model changes related to post-deployment adaptation scenarios.

| Model Element | Kind | Description |
|---|---|---|
| AvailabilityZone.racks | Addition | A rack to be integrated into a DC's availability zone |
| Rack.nodes | Addition | (new) node(s) to be integrated into the cloud |
| Rack.nodes | Deletion | (faulty) node(s) to be removed from the cloud |
| Node.nodeType | Modification | Reassignment of a node to another category |
| Node.disks | Addition | (new) disk(s) to be added to a node |
| Node.disks | Deletion | (faulty) disk(s) to be removed from a node |

element and as supported by existing API calls – the change impact and resulting adaptation actions cannot be determined without further domain knowledge. In some cases a change implies a trivial update of a configuration option; in other cases it is not possible at all to apply the described change to an existing IaaS deployment without affecting tenants during production.

Some adaptation scenarios relate to the patch management of an OS as well as SDN, storage, and IaaS solutions. That is, updates such as security patches and upgrades in case of new software releases shall be applied in existing deployments. The version control of respective services accounts for a corresponding view. That is, distinct models hold respective information such as releases and versions.

## 6 The Ming Prototype

For the implementation of the MING prototype, the Eclipse Modeling Framework [51] was chosen as a modeling foundation. Eclipse Xtext [53] (Xtext) served for defining the DSL and its views and for obtaining a respective DSL editor. EMF Compare [52] is used for calculating a diff-model. Finally, model transformations were implemented in Eclipse Xtend [54] (Xtend).

In the following, the engineering process is described that was executed when realizing the MING prototype. Also some excerpts from code generation templates are depicted. Next, examples in form of DSL views demonstrate how the MING prototype is used for specifying different deployment configuration options. Finally, post-deployment scenarios are discussed.

### 6.1 Initial Engineering

Prior to adopting the MBE approach for basing the automation on models a MAAS configuration and OpenStack JuJu bundle were engineered and tested. These files served as the target code and constituted a starting point for the reverse engineering. For this, values that are specific to a DC deployment were identified first. Next, loop statements and rules were introduced for generating repetitive code and for improving its quality. In this process, the target code was transformed to a M2T code generation template. Also, a metamodel was established. For this, a grammar of a DSL was defined with the intention to act as an interface for stakeholders for expressing and representing various aspects related to the deployment of DCs. For supporting SoC, different concepts were separated into distinct views. Next, the configuration options from the target code were sublimated into models. That is, these values were stored in models that conform to the metamodel as used by the M2T transformation. Finally, the original code was produced from the models using code generation. From this point on, it became possible to base the overall deployment on models. Relating to certain views, it also became possible to reuse models easing the deployment of multiple DCs.

Figure 3 depicts an excerpt from the M2T transformation for generating a MAAS configuration. Three FOR loops iterate over all DC's availability zones,

```
nodes:
  «FOR az : dc.azs»
  «FOR rack : az.racks»
  «FOR node : rack.nodes»
  - name: «model.deployment.name»-«node.name»
    «IF node.nodeType == NodeTypeName.CN»
    tags: «getComputeAggregate(zones, node, az)»
    «ELSEIF node.nodeType == NodeTypeName.SN»
    tags: storage-«getCephPool(zones, node).name»
    «ELSEIF node.nodeType == NodeTypeName.MN»
    tags: api
    «ELSEIF node.nodeType == NodeTypeName.NN»
    tags: gateway-«getGatewayZone(zones, node).name»
    «ELSE»
    tags: standby
    «ENDIF»
    architecture: «node.arch»/generic
    mac_addresses:
    «FOR nic : getNICs(node)»
      - «nic.mac»
    «ENDFOR»
    power:
      type: ipmi
      address: «getIPMI(node)»
      user: «model.credentialsIPMI.username»
      pass: «model.credentialsIPMI.password»
      driver: LAN_2_0
    «enrichWithIPs(node)»
    «FOR nic : getNICs(node).filter[it.ip4 != null]»
    sticky ip address:
      mac_address: «nic.mac»
      requested_address: «nic.ip4»
    «ENDFOR»
  «ENDFOR»
  «ENDFOR»
  «ENDFOR»
```

**Fig. 3.** Code Generation for Metal as a Service (MAAS) `nodes` with Xtend.

racks, and finally nodes. As a result an entry is generated for every node containing all of its MAC addresses and assigned IP addresses. The latter are specified in a `sticky_ip_address` section. Code generation assures the consistency between the MAC addresses.

Please note that a separate template that supports a different target technology can process the same models. That is, while the models describe the overall DC deployment, they are agnostic to actual deployment automation technologies.

### 6.2 Continuous Improvements

Yet, following an agile approach, the engineering of the target code was subject to an iterative process. For this reason also the templates, the metamodel, and the models had to be revised repeatedly. For comprising increments such as new features or bugfixes, the YAML files were improved and tested by cloud engineers. When a new stable version of target code became available, a new reverse engineering cycle was started for capturing the respective expert knowledge in model transformations. For this, the differences in the target code were analyzed and the template aligned accordingly. If necessary, new concepts were added to

```
phase2:
  inherits: phase1
  services:
    «FOR cluster : zones.cephClusters»
    ceph-«cluster.name»:
      charm: cs:«deployment.dist.getName()»/ceph
      num units: «cluster.nodes.size»
      options:
        osd-devices: «deployment.ceph.osdDevices»
        osd-reformat: '«IF deployment.ceph.osdReformat»True«ELSE»False«ENDIF»'
        osd-format: '«deployment.ceph.osdFormat.literal»'
        ceph-public-network: «getNetwork(networks, 'cephFE' + cluster.name)»
        ceph-cluster-network: «getNetwork(networks, 'cephBE' + cluster.name)»
      to:
        «FOR i : 0..cluster.nodes.size-1»
        - '«deployment.name»-dc-storage-«cluster.name»=«i»'
        «ENDFOR»
    «ENDFOR»
```

**Fig. 4.** Code generation of Ceph clusters in a JuJu bundle with Xtend.

the metamodel and the DSL views. This two-phase procedure with a handover of target code for reverse MBE has the advantage that domain experts can continue to work as usual and do not have to be involved closely into MBE activities which they may not be familiar with.

As an alternative to the reverse engineering, increments were sometimes directly realized in the metamodel and the templates. This way, complex changes can sometimes be addressed in a more efficient manner. Generally, however, this requires a close collaboration between MBE and domain experts or either expert knowledge in the domain on part of an MBE expert or a fair understanding of a templating language such as Xtend on part of the domain experts. A change that was easier to realize because of resulting repetitive code was the assignment of nodes to some Ceph clusters or host aggregates. Features such as modeling support for more than one Ceph cluster in a DC were directly realized using loop statements within the transformation template (see Fig. 4).

### 6.3 Example Configuration and Version Control

For configuring OpenStack various variables exist for the different services. Using the DSL editor a user profits from code completion, syntax highlighting, and validation. Figure 5 shows an example configuration view for the deployment of a DC from bare machines with a base OS distribution (`Ubuntu`) and storage (`Ceph`) and IaaS (`OpenStack`) solutions with respective services (e.g., `Neutron` and `Nova`).

Specific versions of the distribution and the various services are configured in a separate view as shown in Fig. 6(a). For example, a certain OS image is referenced for the bare machine installation there. That is, the underscored name is a reference to a definition of the image with metadata such as an Uniform Resource Locator [20], [3] (URL) and checksums.

In addition to the upstream projects, specific artifacts from downstream projects that target specific operating systems or deployment technologies are

```
IaaS Deployment SongThrush @ MingDC9

OperatingSystem Ubuntu

Ceph
      osdFormat btrfs
      osdReformat

OpenStack
      Heat
            workers 8
            hiddenTags "generated"
      Neutron
            externalBridge "br-ext"
            overlayNetType "gre vxlan"
            l3_ha
            l3_agents 2-4
            mtu 9000
      Nova
            liveMigration
            mtu 9000
      Percona
            maxConnections 12345
```

**Fig. 5.** Configuration of an IaaS deployment – DSL view.

recorded in another view. For the Ceph and OpenStack services, Fig. 6(b) shows versions of Debian packages as found in the Ubuntu Cloud Archive [9] that relate to respective upstream projects and versions. The DSL (editor) facilitates consistency between the versions of the up- and downstream projects through validation and scoping while offering a selection of existing matching packages in code completion.

Although these views exist, it is not compulsory to specify certain versions. That is, it is possible to use default versions in a deployment. Yet, at a certain point in time it is important to fix and freeze the set of versions of the various projects. Ensuring the roll-out of defined software artifacts, this permits for reproducibility of deployments in particular. In addition, it facilitates patch management in a production environment.

### 6.4 Code Generation

Together with the other views (e.g., an instance of the metamodel as shown in Fig. 1) and the default models all required information is complete for model transformation to take place. The current prototype supports generation of a MAAS configuration and a OpenStack JuJu bundle in form of YAML files. For the sublimated configuration using default options and versions, the ratio between the size of Ming models and YAML code for MAAS and JuJu yielded 27%. That is, the models in Ming are nearly four times more compact than the corresponding code as generated by the templates.

```
Upstream Project Versions
SongThrush @ MingDC9

Ubuntu 16.04.1 LTS (Xenial Xerus)

Ceph v10.2.0 (Jewel)

OpenStack 2016-04 (Mitaka)
      Ceilometer "6.0.0"
      Cinder "8.0.0"
      Glance "12.0.0"
      Heat "6.0.0"
      Horizon "9.0.1"
      Keystone "9.0.0"
      Neutron "8.1.2"
      Nova "13.0.0"
```

(a) Versions of IaaS Upstream Projects

```
Package Versions
SongThrush @ MingDC9

Ceph "10.2.0-0ubuntu0.16.04.2"

OpenStack
      Ceilometer "1:6.0.0-0ubuntu1"
      Cinder "2:8.0.0-0ubuntu1"
      Glance "2:12.0.0-0ubuntu2"
      Heat "1:6.0.0-0ubuntu1.1"
      Horizon "2:9.0.1-0ubuntu2"
      Keystone "2:9.0.0-0ubuntu1"
      Neutron "2:8.1.2-0ubuntu1"
      Nova "2:13.0.0-0ubuntu5"
```

(b) OS Packages of IaaS Solutions

**Fig. 6.** Version control – DSL views.

### 6.5 Post-Deployment Adaptation Scenarios

As described, the MING approach provides a framework for describing the various aspects of DC deployments. Relying on existing deployment tools, this permits automation from a modeling level. Beyond such deployment automation MING can help to facilitate operational scenarios as well. For this, models relating to a deployment are modified by DSL users and are analyzed in MING in a first step. This requires that the models are put under version control, e.g., using Git [30], and are up to date, i.e., truthfully reflect the DC deployment.

Based on the same MING metamodel, model differences between two model versions are calculated using EMF Compare. In case the types of changes are supported for automation, a respective adaptation process can be initiated. Besides, MING can help experts to describe and learn about the change impact and best practices also including other types of modifications. That is, documentation can be written for different types of modifications that will be looked up and presented to the experts. This way, MING can be used to interactively try out modifications, learn or document knowledge related to the changes, and, if available, trigger an adaptation process for enforcing the changes.

**Scaling.** One of the adaptation scenarios in a production environment relates to the scaling of a DC as described in Sect. 2.4. For this new racks with new nodes are added to an inventory model of the DC. There, nodes can be assigned a `nodeType`. Besides, nodes can be dedicated to some aggregate.

Figure 7(a) shows an example for an inventory DSL view of the `MingDC9`. A new rack (`r9`) is added to one of the availability zones (`az1`). It contains a `storage` node (`n8`) with a couple of NICs and disks. The DSL view in Fig. 7(b)

```
DataCenter MingDC9

AvailabilityZone "az1"

Rack "r9"

Node "n8" type storage
      NIC network ipmi id 23
      NIC "01:02:03:04:05:06" network oam id 45
      NIC "07:08:09:0a:0b:0c"
      NIC "0d:0e:0f:10:11:12"
      HDD 2 TB
      HDD 2 TB
      HDD 2 TB
      HDD 2 TB
      HDD 2 TB
      SSD 512 GB
```

(a) Inventory

```
Aggregates for MingDC9

CephCluster "public"
"MingDC9.az1.r1.n3"
"MingDC9.az2.r2.n4"

Aggregate "EPA"
"MingDC9.az1.r1.n5"
"MingDC9.az2.r2.n6"
```

(b) Node Assignments

**Fig. 7.** Inventory and node assignments (excerpts) – DSL views.

illustrates an example with an excerpt referencing other nodes from the inventory model. It defines a Ceph cluster (`public`) and an host aggregate (`EPA`) with two nodes respectively (`n3, n4` and `n5, n6`). These nodes are located in two racks (`r1` and `r2`) in different availability zones (`az1` and `az2`). Part of the inventory, node `n8` can now be added to the Ceph cluster `public`. That is, a reference (`"MingDC9.az1.r9.n8"`) can be added to the list. After having finished the modifications, the DSL user may now start an adaptation analysis. For this, the model differences (i.e., a new rack within the inventory and a new node in the Ceph cluster) are identified based on the previous version of the MING views. Next, the user receives previously documented information that describes the adaptation process that has been automated and can be triggered. For enforcing the adaptation, the new node is installed with an OS (see also Figs. 5 and 6(a)), a Ceph service (see Fig. 6(b)), and integrated into the Ceph cluster.

**Patch Management.** Another use case is support for security patches and software updates as pointed out in Sect. 2.3. For this, the responsible DSL views are related to the version control of software projects and artifacts (see Fig. 6). In the DSL editor new available versions (e.g., Charms and/or (related) OS packages) can be highlighted to the user for selection. For enforcing the changes, an adaptation process may possibly rely on the (re)generation and deployment of a JuJu bundle if supported by the respective Charms.

## 7 Discussion

The model-based approach enables the utilization of diverse technologies. In this work, MAAS and JuJu constituted target technologies. For supporting other deployment tools, the process described in the Sects. 6.1 and 6.2 can be used. Given availability of respective templates, this enables evaluation of different

deployment tools. That is, from the same models target code is generated for various deployment tools using respective M2T transformations. This in turn fosters a common datamodel for establishing a standard for configuring an IaaS deployment from bare machines. With the separation of the models in distinct views further benefits can be leveraged:

Discovery of nodes and their components automatically yields a certain view. Credentials as stored in another view are generated initially if not set for a certain deployment. In both cases parts of the overall configuration are provided and the DSL user only needs to focus on the other aspects of a deployment.

For a given DC it is possible to specify different deployments. That is, while the physical setup (i.e., availability zones, racks, and nodes) as captured in one view does not change, views related to the deployment such as the assignment of nodes or the deployment and configuration of OpenStack services may be different. Yet, only a part of the overall configuration is adapted and existing views can be reused avoiding software clones. This way, deployments can be tested and the differences between them can be described precisely by comparing two models.

In case a different DC shall be deployed similarly, it is possible to reuse views such as the configuration of OpenStack services. This eases the deployment of multiple DCs with a tested configuration.

The possibility to specify default configuration options in models permits custom user-defined defaults. The fact that these are then applied on the views has two major advantages: It simplifies the models by moving default configuration options out of the views making the files more compact. Also, it simplifies code generation by only processing the resulting model and makes it independent from any (user-defined) defaults.

Regardless of the benefits of this model-based approach, it is always possible to continue work with the output. Thus, MING does not introduce any dependency in regard to the underlying automation.

Not all fine-grained configuration options of the bare machine provisioning or IaaS solutions are reflected in the MING metamodel. Thus, in case these shall be lifted to the modeling layer, the metamodel and the views need to be extended. In case multiple technologies are supported using code generation, certain features of one technology may not be supported by alternatives. For example, the deployment of IaaS services may be realized using Kernel-based Virtual-Machine (KVM), Linux Containers (LXC), or Docker. In case such a configuration option is specified but not supported by a technology a fallback may be realized during code generation. For early feedback this can be addressed through validators in the DSL. That is, when selecting an option that is not supported by some technologies a warning is displayed in the DSL editor.

As pointed out in Sect. 5.2 adaptation actions can generally be derived from inventory changes. The change impact of some other model differences needs to be examined case by case. For this, respective actions can be identified, documented, and implemented if possible. Although arduous, such an endeavor certainly is worthwhile as it realizes operational support for different scenarios as

described from a technology agnostic modeling level. In this regard it should be pointed out that – even in production – not all possible model changes need to be covered necessarily. In the course of an agile development, different scenarios could be formulated as user stories and be part of a prioritized (Scrum [46]) backlog.

There is a need to keep the models up to date. That is, changes that are performed in a datacenter (hard- and software) also need to be reflected in the models. Ideally, the models would be in sync with reality; furthermore, they would be causally connected. That is, model changes would imply appropriate adaptations. For this, MING could be extended with a dedicated models@runtime layer (cf. [6]).

In addition to deployment and adaptation the model-based framework could cover monitoring aspects as well. This would further strengthen MING from an operational point of view. Relating to, e.g., host aggregates that run low on resources (see also Sect. 5.2) an adaptation process could automatically be triggered for balancing out resources.

## 8 Conclusion

Gaining independence from existing deployment technologies MING establishes a metamodel that serves as a technology agnostic datamodel for the configuration of IaaS deployments in DCs from bare metal. The model-based approach integrates with available deployment tools and realizes SoC through view models backed by a textual DSL. For facilitating operational aspects posterior to initial deployments, MING analyses changes on a modeling level. It supports experts to both document and learn about different types of adaptations and their change impact and can, if available, trigger an appropriate process.

**Acknowledgments.** The author would like to thank the members of the extended Infrastructure Cloud team, i.e., Alexandros Tsirepas, Andreas Flick, Axel Clauberg, Basil Ahmed, Bernard Tsai, Daniel Brower, George Wu, Herbert Damker, Karsten Reincke, Ken Jung, Matthias Britsch, Michael Linke, Michael Machado, Normen Kowalewski, Patrick Münch, Rainer Schatzmayr, Robert Schwegler, Seth Chen, Stefan Schraub, Steve Liu, Thomas Hillen, Thomas Oswald, Tobias Brausen, and Tomislav Sukser for their dedicated endeavors making this work possible, valuable feedback, and helpful comments.

## References


1. Alizadeh, M., Edsall, T.: On the data path performance of leaf-spine datacenter fabrics. In: IEEE 21st Annual Symposium on High-Performance Interconnects, HOTI 2013, Santa Clara, CA, USA, 21–23 August 2013, pp. 71–74. IEEE Computer Society (2013)
2. Andrikopoulos, V., Binz, T., Leymann, F., Strauch, S.: How to adapt applications for the cloud environment. Computing **95**, 493–535 (2013)
3. Berners-Lee, T., Masinter, L., McCahill, M.: Uniform Resource Locators (URL), December 1994. http://ietf.org/rfc/rfc1738.txt. Accessed Sept 2016
4. Bernstein, D.: Containers and cloud: from LXC to Docker to Kubernetes. IEEE Cloud Comput. **1**(3), 81–84 (2014)
5. Bézivin, J.: On the unification power of models. Soft. Syst. Model. **4**(2), 171–188 (2005)
6. Blair, G.S., Bencomo, N., France, R.B.: Models@ run.time. Computer **42**(10), 22–27 (2009). IEEE
7. Canonical, Ltd.: JuJu. http://jujucharms.com. Accessed Sept 2016
8. Canonical, Ltd.: MAAS: Metal as a Service. http://maas.io. Accessed Sept 2016
9. Canonical, Ltd.: Ubuntu Cloud Archive. https://wiki.ubuntu.com/OpenStack/CloudArchive. Accessed Sept 2016
10. Cerf, V.G., Khan, R.E.: A protocol for packet network intercommunication. IEEE Trans. Commun. **22**, 637–648 (1974)
11. Chandrasekar, A., Gibson, G.: A comparative study of baremetal provisioning frameworks. Technical report CMU-PDL-14-109, Parallel Data Lab, Carnegie Mellon University, December 2014. http://pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-14-109_abs.shtml. Accessed Sept 2016
12. Chef Software, Inc.: Chef. http://getchef.com. Accessed Sept 2016
13. DeHaan, M.: Cobbler. http://cobbler.github.io. Accessed Sept 2016
14. DeHaan, M.: Ansible (2012). http://ansible.com. Accessed Sept 2016
15. Dell. Inc.: Crowbar. http://crowbar.github.io. Accessed Sept 2016
16. Docker, Inc.: Docker (2013). http://docker.com. Accessed Sept 2016
17. Droms, R.: Dynamic Host Configuration Protocol. RFC 2131, The Internet Engineering Task Force, March 1997. http://ietf.org/rfc/rfc2131.txt. Accessed Sept 2016
18. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns - Fundamentals to Design, Build, and Manage Cloud Applications. Springer, Wien (2014)
19. Gärtner, M., Lange, T., Rühmkorf, J.: The fully automatic installation of a Linux cluster. Technical report 379, Computer Science Department, University of Cologne, December 1999. http://e-archive.informatik.uni-koeln.de/id/eprint/379. Accessed Sept 2016
20. Hansen, T., Hardie, T., Masinter, L.: Guidelines and Registration Procedures for New URI Scheme, February 2006. http://ietf.org/rfc/rfc4395.txt. Accessed Sept 2016
21. Hansson, D.H.: Ruby on Rails (2005). http://rubyonrails.org. Accessed Sept 2016
22. Holmes, T.: Facilitating migration of cloud infrastructure services: a model-based approach. In: Paige, R.F., Cabot, J., Brambilla, M., Hill, J.H. (eds.) Proceedings of the 3rd International Workshop on Model-Driven Engineering on and for the Cloud co-located with the 18th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, Canada, 29 September 2015. CEUR Workshop Proceedings, vol. 1563, pp. 7–12. CEUR-WS.org (2015)

23. Holmes, T.: Sublimated configuration of infrastructure as a service deployments – Ming: a model- and view-based approach for cloud datacenters. In: Cardoso, J., Ferguson, D., Muñoz, V.M., Helfert, M. (eds.) 6th International Conference on Cloud Computing and Services Science. vol. 2, pp. 308–313. SciTePress (2016)
24. Intel Corporation: Preboot Execution Environment (PXE) Specification Version 2.1, September 1999. http://download.intel.com/design/archives/wfm/downloads/pxespec.pdf. Accessed Sept 2016
25. Intel Corporation, Hewlett-Packard Company, N.E.C., Corporation, Dell Inc.: Intelligent Platform Management Interface Specification v2.0 rev. 1.1, October 2013. https://www-ssl.intel.com/content/www/us/en/servers/ipmi/ipmi-second-gen-interface-spec-v2-rev1-1.html. Accessed Sept 2016
26. International Organization for Standardization: ISO/IEC 42010:2011 Systems and software engineering - Architecture description, December 2011. http://iso.org/iso/catalogue_detail.htm?csnumber=50508. Accessed Sept 2016
27. Kelly, P., Levy, O.: Foreman (2009). http://theforeman.org. Accessed Sept 2016
28. Lange, T.: Fully Automatic Installation (2000). http://fai-project.org. Accessed Sept 2016
29. Lange, T.: 10 Jahre FAI Projekt. Technical report 603, Computer Science Department, University of Cologne, July 2010. http://e-archive.informatik.uni-koeln.de/id/eprint/603. Accessed Sept 2016
30. Torvalds, L.: Git, April 2005. http://git-scm.com. Accessed Sept 2016
31. Maggiani, R.: Cloud computing is changing how we communicate. In: International Professional Communication Conference, pp. 1–4 (2009)
32. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G.M., Peterson, L.L., Rexford, J., Shenker, S., Turner, J.S.: Openflow: enabling innovation in campus networks. Comput. Commun. Rev. **38**(2), 69–74 (2008)
33. Mell, P.M., Grance, T.: The NIST definition of cloud computing. Technical report, SP 800-145, National Institute of Standards & Technology (2011)
34. Mens, T., Gorp, P.V.: A taxonomy of model transformation. Electr. Notes Theor. Comput. Sci. **152**, 125–142 (2006)
35. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4), 316–344 (2005)
36. Miller, J., Mukerji, J.: MDA Guide Version 1.0.1. http://omg.org/cgi-bin/doc?omg/03-06-01. Accessed Sept 2016
37. OpenStack Foundation: Compass. http://syscompass.org. Accessed Sept 2016
38. OpenStack Foundation: Fuel. http://wiki.openstack.org/Fuel. Accessed Sept 2016
39. OpenStack Foundation: OpenStack Bare Metal Provisioning (Ironic). http://wiki.openstack.org/Ironic. Accessed Sept 2016
40. Openstack Foundation: OpenStack on OpenStack (TripleO). http://wiki.openstack.org/TripleO. Accessed Sept 2016
41. OpenStack Foundation: Packstack. http://wiki.openstack.org/Packstack. Accessed Sept 2016
42. OpenStack Foundation: OpenStack, July 2010. http://openstack.org. Accessed Sept 2016
43. Puppet Labs, L.L.C.: Puppet. http://puppetlabs.com. Accessed Sept 2016
44. Red Hat, Inc.: Kickstart (2011). http://github.com/rhinstaller/pykickstart. Accessed Sept 2016
45. Red Hat, Inc.: RPM Distribution of OpenStack (RDO) (2013). http://rdoproject.org. Accessed Sept 2016
46. Schwaber, K., Beedle, M.: Agile Software Development with Scrum, 1st edn. Prentice Hall PTR, Upper Saddle River (2001)

47. Software in the Public Interest, Inc.: Debian (1993). http://debian.org. Accessed Sept 2016
48. Sollins, K.R.: The TFTP Protocol (Revision 2). RFC 1350, The Internet Engineering Task Force, July 1992. http://ietf.org/rfc/rfc1350.txt. Accessed Sept 2016
49. The Apache Software Foundation: Hadoop (2011). http://hadoop.apache.org. Accessed Sept 2016
50. The CentOS Project: CentOS (2004). http://centos.org. Accessed Sept 2016
51. The Eclipse Foundation: Eclipse Modeling Framework Project (EMF) (2002). Accessed Sept 2016. http://eclipse.org/modeling/emf
52. The Eclipse Foundation: EMF Compare, October 2006. http://wiki.eclipse.org/EMF_Compare. Accessed Sept 2016
53. The Eclipse Foundation: Xtext (2006). Accessed Sept 2016. http://eclipse.org/Xtext
54. The Eclipse Foundation: Xtend, June 2011. http://eclipse.org/xtend. Accessed Sept 2016
55. Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013)
56. Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E., Maltzahn, C.: Ceph: a scalable, high-performance distributed file system. In: Bershad, B.N., Mogul, J.C. (eds.) 7th Symposium on Operating Systems Design and Implementation, pp. 307–320. USENIX Association (2006)
57. Wettinger, J., Breitenbücher, U., Leymann, F.: Standards-based DevOps automation and integration using TOSCA. In: 7th IEEE/ACM International Conference on Utility and Cloud Computing, pp. 59–68. IEEE (2014)
58. Zimmermann, H.: OSI reference model - the ISO model of architecture for open systems interconnection. IEEE Trans. Commun. **28**(4), 425–432 (1980)