Master Thesis

to obtain the degree “Master of Science”

Extending Infrastructure-as-Code to bare-metal

at the	Ulm University of Applied Sciences Faculty of Computer Science Degree program Intelligent Systems
submitted by	Till Hoffmann
Matriculation number	3135572
for the	Daimler TSS GmbH
Supervisor	Benjamin Gotzes
First advisor	Prof. Dr. rer.nat. Stefan Traub
Second advisor	Prof. Dr.-Ing. Philipp Graf

Submitted on 2021-10-28

Abstract

TBD

The result was published under MIT licence and is available at `https://github.com/thetillhoff/master-thesis/tree/main/src`.

Glossar

YAML YAML is yet another markup language.

Acronyms

AMQP Advanced Message Queuing Protocol

API Application Programming Interface

AWS Amazon Web Services

Azure Microsoft Azure

BIOS basic input/output system

BMC Baseboard Management Controller

BOOTP Bootstrap Protocol

CI/CD Continuous Integration / Continuous Deployment

CNCF Cloud Native Computing Foundation

CSAR Cloud Service ARchive

DHCP Dynamic Host Configuration Protocol

DNS Domain Name System

DSL Domain-Specific Language

FPGA Field-Programmable Gate Array

GCP Google Compute Platform

GPL General-Purpose Language

GRUB GNU GRand Unified Bootloader

HCL HashiCorp Configuration Language

HOT Heat Orchestration Template

HTTP Hyper Text Transfer Protocol

IaaS Infrastructure-as-a-Service

IaC Infrastructure-as-Code

IPMI Intelligent Platform Management Interface

JSON JavaScript Object Notation

KVM Kernel-based Virtual Machine

KVM Keyboard, Video, Mouse

LOM Lights Out Management

MQTT Message Queuing Telemetry Transport

NBP Network Bootstrap Program

NIC Network Interface Card

OASIS Organization for the Advancement of Structured Information Standards

OCCI Open Cloud Computing Interface

OGF Open Grid Forum

OOB Out Of Band Management

OS Operating System

PaaS Platform-as-a-Service

PXE Preboot eXecution Environment

RPC Remote Procedure Call

SaaS Software-as-a-Service

SAML Security Assertion Markup Language

SSH Secure Shell

TFTP Trivial File Transfer Protocol

TOSCA Topology and Orchestration Specification for Cloud Applications

UEFI Unified Extensible Firmware Interface

VM Virtual Machine

WOL Wake On LAN

YAML Yet Another Markup Language

Table of contents

1 Introduction

Todays distributed applications don't scale in the range of tens or hundreds of nodes but in tens of thousands [1], [2] [3]. In order to be as fast and efficient as possible, the number of nodes has to automatically scale up and down based on their usage. The conditions are simple (f.e. "add a node when all nodes have reached 80 percent cpu load") but the frequency for triggers is high. to simplify management of nodes for both the cluster software and administrations, each node should also be set up the same way. A perfect use-case for automation.
The process of software-defining infrastructure is called Infrastructure-as-Code (IaC). Using software development tools to manage infrastructure has many more advantages like version-control, collaboration, reviews, automated tests and continuous deployment. The accompanying combination of development and operations called DevOps opened a complete new field in computer science [4]. To be able to increase the amount of components than can be software-defined, the underlying hardware needs to support it. As an example, processors have a fixed architecture, while with Field-Programmable Gate Array (FPGA) chips it is configurable.

Such hardware features are exposed via a corresponding Application Programming Interface (API). Some hardware properties cannot be changed via software, for example how many physical machines exist in a certain environment. A partial solution for such cases are abstraction layers like virtualization.
But virtualization only provides an API on a single host; In order to be scalable and in order to be able to distribute new workloads in the most efficient way (f.e. putting a new Virtual Machine (VM) on the hypervisor with the lowest load), an orchestrating software is needed. Examples for such software are VMware's vSphere, OpenStack but also Google's Kubernetes.
These tools are capable of automatic live-migrations of workloads in order to distribute load more equally, and provide APIs for their features.
Another category for such orchestrators are public cloud providers like Amazon's Web Services, Microsoft's Azure and Googles Cloud Platform. They as well provide APIs for their features.
While these API-providers allow their users to have a simplified view on provisioning, they just shift the effort of managing the underlying hardware from application developers to the provider software. It is now in the area of responsiblity of the developer team of the latter to manage the underlying hardware (i.e. adding new physical machines to the cluster).
This approach has three main issues: For one, application developers have a hard time switching between or even mixing those providers, since their APIs are very different. Second, these orchestrators all do mostly the same thing, but with different efficiency and flexibility. Third, each one of them has one initial requirement: Someone has to do the initial bootstrapping, i.e. somehow set up the orchestrator. Again, this does not solve the hardware management problem, but shifts it to

a different problem which (hopefully) requires less effort to solve.
This thesis aims at three fundamental questions: Can bare-metal machines be deployed on-demand like virtual workloads on providers. Is it possible to do so without the requirement of an always-on operator, thus removing the initial bootstrapping effort. And last but not least, how can hardware constraints be mirrored in IaC languages.
The paper at hand first explains how workload provisioning historically evolved and introduce terms required to understand the topic. Then it describes the current state of the art of IaC and provisioning in order to identify issues and where compatiblity makes the most sense. Afterwards different languages to describe IaC will be compared and the most fitting one selected. Before the architecture of an example tool can be discussed, the final constraints and goals for it will be determined. A final discussion analyses the results and answer the initial questions.

2 Background

Searching online for IaC quickly leads to the terms such as “snowflake”, “pet” or “cattle”. [5] In this context, the former two are synonyms and refer to directly/manually managed (configured and maintained) machines. Typically, they are unique, can never be down and ”hand fed” - it is not feasable to redeploy them [6]. The latter is used when referring to machines, that are never directly interacted with; All administrative interactions with them are automated. The approach of treating machines as cattle aims to unify and therefore reduce the administration effort for large amounts of servers. When operating on such larger scales, it is easier to maintain some kind of automation framework and unify the deployment of machines than to administrate each server manually. At the same time, cattle-machines are replacable by design, which is not the case for pet-machines. But even before those terms were introduced, some datacenters were already too large to maintain each server manually. This chapter will guide through a part of history of datacenter technologies, explain how they work whenever they are necessary to understand the further chapters and identify their primary issues.

2.1 Bare-metal

In the early times of datacenters, they required quite the administrative effort. Reinstalling an operating system on a server required one administrator to be physically located close to the server, some kind of installation media, a monitor and at least a keyboard. Since both monitor and keyboard were rarely used, Keyboard, Video, Mouse (KVM) (not to be confused with the linux kernel virtual machine with the same abbreviation) quickly gained foothold. KVM had one set of IO-devices like monitor and keyboard attached on one side and several servers on the other side. Pressing a corresponding button, the complete set of IO-devices would be “automatically” detached from whatever server it was previously connected to and attached to the machine the button refers to.
Those devices still exist and evolved into network-attached versions, which means they don’t require administrators to press buttons on the device and instead of dedicated set of IO-devices per handful of servers, they allow administrators to use the ones attached to their workstation. So these devices introduce some kind of remote control for servers, including visual feedback. Their main issue is not the dedicated cabling they require to each server, but the limited amount of servers they can be attached to. The largest KVM-Switches have 64 ports [7], meaning they can be attached to 64 machines. For datacenters with more machines, this type of management doesn’t scale very well (even financially, since those 64-port switches tend to cost as much as a new car).
Instead of installing each operating system manually, two methods for unattended

installations emerged: One is the creation of so-called “golden images”, where all needed software is preinstalled, settings are baked in, correct drivers are in place and so on [8]. The other is closely related and has a different name for each operating system. Examples are “preseed” for debian, “setupconfig” for windows, “cloud-init” for various operating systems including ubuntu [9]. Under the hood they all work the same: Instead of asking the user each question during setup, the answers are predefined in a special file. This file can be baked in into the golden image or seperately (even on-demand via network).
With those methods, administrators only need to attach the installation medium, configure the machine to boot from it and power-on the machine. While this does save a large amount of time already, it still requires manual interactions with the machine.
To further automate machine installations, technologies like Trivial File Transfer Protocol (TFTP) (1981), Preboot eXecution Environment (PXE) (1984), Bootstrap Protocol (BOOTP) (1985) emerged and concluded in the development of Dynamic Host Configuration Protocol (DHCP) (1993). Only when Intel released the Wake On LAN (WOL) in 1997 and PXE 2.0 as part of its Wired-for-Management system in 1998 it was possible to fully network-boot a device.
PXE uses DHCP to assign an ip-address to a Network Interface Card (NIC). When the NIC receives a so-called “magic packet” during the WOL process, it triggers the machine to power-on. Depending on the basic input/output system (BIOS) or Unified Extensible Firmware Interface (UEFI) (which is a newer implementation of the former) settings, the machine starts with its configured boot-order, for network-boot this means an embedded Network Bootstrap Program (NBP) (for example the widespread network boot loaders PXELINUX or iPXE), which are like a networking equivalent to what GNU GRand Unified Bootloader (GRUB) and the Windows Boot Loader are for local disks: It downloads a kernel from a (network) resource, loads it into memory and finally (chain-)boots the actual Operating System (OS) [10] [11].
The combination of all those technologies finally allows to remotely power-on a machine, boot a kernel via network instead of a local disk, and using the NIC as the interface for those abilities, outsourcing the bootstrapping and scaling to the network infrastructure.
But there are still some issues with those technologies:
When a machine had an error which made it unresponsible for remote access (like SSH), but didn’t power the machine down neither, again an administrator was required to phyiscally attend the server and manually resolve the issue.
The next generation of servers (since 1998) had such a remote control integrated into their mainboard, rendering KVM obsolete, because this new method scales vertical: Every new server has an embedded chip that acts as an integrated remote control. Unifying those efforts into a single standard for the whole industry, Intel published a specification called Intelligent Platform Management Interface (IPMI) around that. Instead of “only” the ability of remote-controlling a server with keyboard, mouse and monitor, IPMI allows administrators to mount ISO images remotely (in a way like network-boot, but a different approach), change the boot order, read hardware sensor values during both power-on- and -off-times and even

control the power-state of the machine. Especially the last part now allows administrators to maintain serves completely remotely via network, making physical attendance only required for changing physical parts of the intrastructure. The aforementioned embedded chips are called Baseboard Management Controller (BMC) and the surrounding technology is called Out Of Band Management (OOB) or Lights Out Management (LOM). Even though these are universal terms for the chips and the technology, most hardware manufacturers have their own name for their specific toolset. Examples are (no need to remember for this thesis) DRAC from DELL EMC, ILO from HPE and IMM from IBM. Not only their names are different: Many features have different names but are actually doing the same. Probably due to their original purpose, those chips are not embedded in every modern mainboard, but only available in server- and enterprise-desktop-mainboards.
There are two different sets of problems solved with all those technologies: The combination of IPMI and LOM allows administrators to debug a machine even on the other side of the planet by giving them remote input/output capabilities as well as power cycle control and the ability to get hardware sensor information. But LOM is almost impossible to automate. The network-boot technologies around PXE on the other side help with automating a high number of servers in parallel, but don't really help with debugging errors.
Together, these solutions enable administrators to automate hardware provisioning at scale while at the same time providing them with remote low-level debugging tools.
These standards are to state-of-the-art remote-server-administration-tools now for several years, along with Secure Shell (SSH). They mostly solve the administration scaling problem and form the base for other tools (more on them later).
Sometimes, it is necessary to power a machine down. Be it for exchanging/adding hardware components or other maintenance. Therefore a best-practice seperates different workloads on different machines. This has the advantage that f.e. powering down a web-server, doesn't impact a database-server. At the same time it has the downside that servers are not efficiently used: When the database has almost no load, but the web-server is close to its limit, load cannot be distributed between them. This is where virtualization comes in.

2.2 Virtualization

Even though IBM shipped its first production computer system capable of full virtualization in 1966 [12], it still took several decades until the "official" break-though of virtualization technologies. Only then were machines powerful enough for virtualization that makes sense in terms of performance, leading to lower management overhead, fewer unused system resources and therefore overall cost savings [13]. Starting 2005, Intel and AMD added hardware virtualization to their processors and the Xen hypervisor was published. Other hypervisors followed: Microsoft Hyper-V and Proxmox Virtual Environment were both published in 2008. The initial release of VMwares ESX hypervisor even dates back to 2001, but evolved to ESXi in 2004.

The first version of the linux kernel containing the Kernel-based Virtual Machine (KVM) hypervisor (not to be mistaken with the equal abbreviation for keyboard, video, mouse described earlier - from this point onwards, KVM always refers to the hypervisor) was published in 2007.
Apart from the previously stated advantages, virtualization allowed for live-migrations of machines to another host without downtime, finally allowing to evacuate a machine prior to maintenance work. The same feature also drastically improves disaster recovery capabilities [14].
But the use of hypervisors and clustering them for live-migration and other cross-node functionalities has a downside as well: Vendor lock-in, since the different VM formats are not compatible (there are some migration/translation tools, but best practices for production environments advise against them), licence / support fees in addition to the hardware support fees and requiring additional expertise for the management software.
Yet, 100 percent of the fortune 500 and 92 percent of all business used (server-)virtualization technologies in 2019 [15] [16] [17].

2.3 Cloud

The term cloud describes a group of servers, that are accessed over the internet and the software and databases that runs on those servers [18]. These servers are located in one or multiple datacenters. There are three types of clouds: Private clouds, which refers to servers and services which are only available internally (i.e. only shared within the organization). The second type are public clouds, which refers to publicly available services (i.e. shared with other organizations) [19]. And lastly, there are hybrid clouds, which mix both of the previous types. All of these have five main attributes in common: They allow for on-demand allocation, self-service interfaces, migration between hosts, as well as replication and scaling of services [lecture notes, VSYS, during bachelor, and [20]].
The public cloud era began with the launch of Amazon's Web Services in 2006. Since then, it evolved into one of the biggest markets with a yearly capacity of $270 billion and an estimated growth of almost 20 percent [21]. The current value exceeds even the market capitalization of Norway [22]. Considering the amount of revenue generated (at least $40 billion [22], it is obvious why the likes as Microsoft (in 2010) and Google (in 2013) followed Amazon into the cloud market [23].
Cloud computing is able to generate these high rates of revenue because they take advantage of economy of scale, very efficient sharing of resources, as well as a combination of a huge amount of developer effort into a low amount of features (in contrast to every organization implementing the same featureset over and over for themselves) [24].
Apart from financial and developer efficiency, clouds have a long list of advantages and disadvantages [Domain-specific language for infrastructure as code].
The high degree of automation and possibilities for scaling within a cloud made it possible to scale automatically. The time required to provision (and deprovision)

new nodes plays an important role for autoscaling. This is where containers come in.

2.4 Containers

While the idea of containers exists for quite some time already (2006 as cgroups, 2007 with LXC, [25] [26], it only reached mainstream popularity with the release of docker in 2013 [27]. The main difference between a VM and a container is the kernel: The former has its own dedicated kernel, which runs in parallel with the hypervisors kernel (yet controlled by it). The latter however shares the kernel of the underlying operating system, thus not requiring a kernel to be loaded for each new instance. As a result, the provisioning speed is dramatically reduced: While VMs are not uncommon to exceed 60 seconds until being fully available, containers only require the time the operating system needs to start a new process, which is sub-second in most cases [28].
Containers also (almost completely) solve the "works on my machine" syndrome, where the developer machine is different to (f.e.) the production system to the extend that a new feature might only work on either, but not both.
Some go even as far as saying containers are the future of cloud computing [29] [30] [31] [32] (or maybe the future of container computing looks different then previously thought [33] [34]).
Docker Inc. also introduced a cross-machine management tool called swarm, which allows users to describe a desired state, which the engine tries to realize (at all times). It was accompanied by Google's Kubernetes in 2014 on the short list of container orchestrators. Kubernetes is based on another (internal) software by Google called Borg, which is the underlying system for software like YouTube, Gmail, Google Docs and their web search. The company had no place to put the open source software, so they partnered with the Linux Foundation to create the Cloud Native Computing Foundation (CNCF) [35]. The CNCF Landscape has since evolved into a multi-trillion dollar ecosystem, so the Kubernetes story only scrapes its surface. The cloud native world has even been labeled as Cloud 2.0 [31].
These orchestrators like Swarm and Kubernetes, along with the cloud providers become more complex with the more features they get, and since the high amount of automation leads to an ever-changing state, several ways to describe the desired state were developed.

2.5 Infrastructure-as-Code

IaC takes advantage of multiple factors:

- Software development encompasses more than running it, f.e. a build pipeline, testing and compliance. All of this has to be documented.

- Documentation is hard to hold up to date [36] [37]. This is not special to orchestrators or cloud providers, but is true for all software.
- The only source of information that cannot lie (i.e. be out of date) is the sourcecode.
- Scaling (infrastructure) leads to standardized objects.
- In order to have multiple instances of the same type of nodes, they have to be provisioned exactly the same.
- The only (reliable) way to something the same way over and over is to script/program them.
- Infrastructure becomes more and more software defined, reducing required physical changes required for changes in the infrastructure (which enables automation).
- Version-control-systems like git are well established and allow for rollbacks, collaboration, reviews and actionability [38]. This improves the quality and enables further automation.

The practice of IaC is best described as finding a compromise between human- and machine-readable languages to describe and directly manage the infrastructure.
Due to the trend towards software-defined everything [39] [40], the advantages gained by using IaC grow steadily. As soon as a software has an API, it can be integrated into IaC. Since the created code only describes how and when to interact with which API and not the actual implementation behind it, some kind of orchestrator is required which processes the requests and runs the actual workflows behind the endpoints.
There are two ways to implement those workflows. The first is a push-based mechanism, where the orchestrator triggers actions on other parts of the system (f.e. commands a hypervisor to create a VM). The other is a pull-based mechanism, where those subsystems (i.e. a hypervisor) periodically asks the orchestrator whether tasks have to be completed [41].
These mechanism not only apply to the interaction between the orchestrator and the subsystems, but between the source and the orchestrator as well.
In order to increase the capabilities of the orchestrator or in other words enable more things to get defined via software, middle- or abstraction-layers are introduced. An example for this is the hypervisor that acts as API-gateway between hard- and software-defined machines. The deployment (and configuration) of that middleware (i.e. the hypervisor) is not within the scope of most IaC frameworks and is outsourced. This layer must be as easy to deploy as possible, making it hard to bring in mistakes and staying as flexible as possible for further configuration via software.
It is obvious, that not everything can be software-defined, since some physical objects (like cables) have to be physically placed [42]. Robots could possibly be used, but in most cases, this is something human workers do. Whether the configuration is correct can often be detected/measured from software. On the other hand, technologies like FPGAs can even change the CPU architecture via software - so

the future might have some surprises in store.
One of the hardest things about applying IaC to bare metal is the complex management and interactions between the multiple APIs. On one side are the "external" protocols and interfaces like DHCP, TFTP, Hyper Text Transfer Protocol (HTTP), Domain Name System (DNS) and SSH. On the other side are the OSs and the features they provide for automation [42]. These range from being able to install the OS in an unattended way, over scriptable settings (or better: The non-scriptable ones - looking at you Windows) to compatibility with widespread instance initialization methods like cloud-init [43].
Another major difference on bare-metal are firmwares. Since they dictate the available version of the hardware APIs, it is important to have them in the correct version [42].

2.6 Domain-Specific Language

As described in the previous chapter, IaC requires an equally machine- and human-readable language. These modeling languages can best be described as Domain-Specific Language (DSL)s as their only purpose is to describe very specific things [44]. Even among those DSLs the domains they can (and want) describe varies a lot. Additionaly, they differ in several properties, for example whether they are graphical or textual; But since IaC is by definition "as code", and code is text-based, corresponding DSLs have to be text-based as well. Examples for well-known DSLs in other domains are SQL and CSS [45]. Another property is the approach, which can be imperative or declarative; Imperative languages describe actions to be done, for example "create X additional instances of Y", whereas declarative languages are used to describe the desired state "I want X instances of Y". When using the latter, it is the orchestrators job to compare the current state against the described desired state and conclude the required actions themselves [46]. Because IaC always aims at describing the whole state, declarative languages are better fitted for this task [38]. They also have the property of being idempotent: If applied multiple times, the result won't change [38]. In order to describe the state of infrastructure, the declarative way is also more intuitive. It is the same way humans would describe a state (i.e. "I see three apples" instead of three times "I see an(other) apple").
Some DSLs (called "internal") in this field are based on another language as XML, JSON, or YAML [38]. This includes both sub- and supersets of them. Libraries are internal DSLs as well lecture notes MODE. "External" DSLs on the other hand are not directly related to other languages lecture notes MODE. An example is the HashiCorp Configuration Language (HCL) used by Terraform [38] lecture notes MODE.
An additional difference between the tools and languages is how they are applied. Some use a push-based mechanism, where f.e. the orchestrator initiates communication with nodes and applies changes. Others use a pull-based mechanism, where the nodes need to watch the(ir) configuration at the orchestrator level and

execute the required actions locally so they become configured as intended. The design decision of push or pull applies to other things as well: How code changes are loaded into the orchestrator for example.
In contrast to a General-Purpose Language (GPL), a DSL allows better seperation of infrastructure code from other code [45]. Additionally, they are more context driven, which makes them easier to work with for domain experts and users lecture notes MODE. Their syntax is smaller and well-defined too, which makes them less complex as well.
In an ideal world, a DSL for IaC is not a limitation factor; For example it is not limited to neither full usage of virtualization, containers nor bare-metal. It should support all of those cases and also allow hybrid scenarios. Additionally, it should be able to describe both small and large environments, while the required effort should increase less than linear. Furthermore, an ideal DSL should not lock into a single vendor, but empower migrations and cross-provider scenarios wherever the user sees fit. This includes the licence and owner of the language; It should not be left in the hands of a single organization, but a group (of several organizations/individuals). While a single owning organization tends to reflect itself in the software [47], a group of organizations or a committee can help in finding a much more universal solution. On the other hand, the more stakeholders are involved, the harder a compromise is to find.

3 Related work

Several tools, frameworks and even whole ecosystems have evolved around IaC. This chapter is focused on finding the most common, determining their use-cases and identifying their issues. Additionally, a simple reference infrastructure will be introduced, which must be deployable with the respective tool.

3.1 State-of-the-art automated hardware provisioning

The interest in IaC has been increasing on a steady level over the last years [48].
It is estimated that ninety percent of global enterprises will rely on hybrid cloud by 2022 [49]. It is also estimated that on-premise workloads drop from 59 percent in 2019 to 38 percent in 2021 and workloads on public clouds grow from 23 percent to 35 percent [50].
Instead of updating deployed instances, recreating them ensures all of them are equal [42]. This includes software and firmware upgrades.
Another reason is heterogeneity in systems: Even when using only a single vendor or even a single model, variations occur. Be it that newer models have upgraded firmwares or other "under-the-hand" changes [42].
It is better not to assume certain states, but check them instead. This way, whenever a state is unexpected, the automation can exclude this certain node and tell the responsible humans to check what's wrong. The only reliable source of truth for the current state is the current state itself - not some kind of cached or partial version of it [42].
So far, public cloud providers haven't exactly published how they are provisioning their bare-metal infrastructure.
But there are some hints, as some of those providers have an on-premises or edge product. The Microsoft Azure Stack HCI cluster is such a case. The documentation recommends starters to get machines with the correct drivers and OS preinstalled [51]. Apart from that, they describe additional OS deployment options like using an answer file (unattended installation), network deployment (PXE), System Center Virtual Machine Manager (only for Windows OSs), and even manual provisioning [51]. The preinstalled OS makes the vendor (in this case Microsoft) reponsible for provisioning. So it doesn't solve the problem but shifts it somehwere else. Additionally, it doesn't work in all cases - for example on reinstallations.
While Amazon Web Services Outposts is a similar product, it doesn't allow customers to manage it themselves. Instead Amazon dispatch their own service personnel for every necessary manual task [52] [53].
Google doesn't have a product to bring its whole cloud on-premise or to the edge yet, but only dedicated featuresets like the Google Kubernetes Engine. The company relies on an underlying VMware vSphere environment and therefore out-

sources hardware management [54].
When a company like Google relies on a third-party software it has to be special in some way.

So how does deployment of VMwares vSphere clusters work? The most important thing with vSphere is that it can be deployed as a VM on an ESXi server (since version 7.0 the appliance is the only way - previously a Windows system could be used as well), the hypervisor OS developed by VMware [55] [56] [57]. In other words, vSphere requires (at least one) manually installed ESXi server, which can then host the vSphere Server software, which then in turn has a feature called Auto Deploy [58]. This feature creates a PXE boot infrastructure that requires an external DHCP server [59] [60]. The latter has to be configured to distribute network boot details which point to the preexisting vSphere Server [59]. In order to reduce deployment time, Auto Deploy does not install the ESXi OS on machines, but loads the boot image directly into its memory [61]. This implies that server restarts are equal to redeployments.
Even VMware doesn't seem to bring a TFTP server as part of their software, but describes how to install and configure a third party product themselves [62]. Since a PXE environment consists of at least a DHCP- and TFTP-server (for performance reasons mostly paired with an HTTP-server) as well as the operating system image, the ratio of reliance on third party products is suprising - considering VMware develops most of their software themselves. But when VMware does rely on this deployment approach, it must have proven to be reliable and hold water. Apaches CloudStack supports two hypervisors; For ESXi it recommends also using vSphere, while for XenServer and KVM it does not specify any deployment options - it documentation starts after the hypervisor is installed [63].
One of the most recently published cluster software for bare metal is Googles Anthos. The software and its documentation completely omit the provisioning part up the point where nodes can only be added when they are already accessable via SSH [64].
Common asset management tools like servicenow or i-doit, use providers like vSphere or public clouds for instantiation [65] or don't support hardware provisioning [66].
Other bare-metal lifecycle management tools like Canonical MAAS, Foreman, FOG, FAI, Cobbler, Openstacks Ironic, RackNs Digital Rebar and Equinix Metals Tinkerbell as well as Microsofts System Center Virtual Machine Manager also rely on PXE for automatic OS deployments [67] [68] [69] [70] [71] [72] [73] [74] [75]. Most often, they have the required software (the aforementioned DHCP- and TFTP-server) embedded in some way and only require minor interactions to configure it properly (like setting up the DHCP range).
Only the minority of bare-metal provisioning software uses or at least supports IPMI as tool of the trade. This includes Canonical MAAS (only for power management), OpenStacks Ironic (for power management and sensor data), RackNs Digital Rebar and ispsystems DCImanager [76] [72] [73] [77]. The main problem with using IPMI for provisioning is its vendor-specific implementations. Not only is it not available for all hardware, but different vendors support different features of IPMI - often even with different APIs. A second, but closely related issue is its unavailability for VMs: Most hypervisors don't support IPMI interfaces for virtual machines.

And even if they do (for example via plugins), their documentation is sparse and their development stale [78].
Another reason for no using IPMI is its historically low security. Although most vendors had their own credentials for accessing the management interface, they used the same combination of user and password for all of their devices [79]. With the taking effect of senate bill 327 chapter 886 (1798.91.04) in january 2020, the vendors must now use a unique random password for each machine.
On the other hand, the BMC has capabilities beyond network boot and WOL. For example it allows administrators to debug an unresponsive machine, execute hard resets and change BIOS/UEFI settings remotely. So while IPMI definitely has its own place, it is not the go-to technology for automated provisioning.
Whenever PXE is used for deployments, as a first step an iPXE image is deployed via TFTP. iPXE is best compared to a customizable and very advanced BIOS/EFI but has several advantages over the default ones. For one, it is scriptable [80]. Therefore it is very flexible in its configuration even during its runtime. And it supports loading the actual OS image via HTTP instead of TFTP. Since iPXE is several times smaller and more lightweight than most operating systems, as well as the fact that HTTP is more performant than TFTP and there exist better tools around it, this approach does not only speed up the deployment but makes it more reliable and customizable, too [81] [82] [83] [84].
The previous part of this chapter focused on the technological “infrastructure” aspect of IaC. Neither less important, less complex nor less diverse is the “as code” part.
As long as there are few properties that change, it is absolutely feasable to use command-line arguments to describe the desired state for IaC tools [38]. But with a growing amount of properties the statespace increases, requiring a better way to describe it: Configuration files. The languages used within those files are mostly DSLs. In contrast to a GPL (not to be confused with the licence), its domain-specific counterpart promises higher success rates even with less experience and significantly higher closeness of mapping [85]. Especially the last attribute helps developers to simplify their state descriptions. Another major advantage of using a GPL is the ecosystem of tools; Because they are well supported by IDEs, they have powerful features like syntax highlighting, code refactoring and testing support [38].

3.2 Domain-Specific Languages for Infrastructure-as-Code

There is a vast amount of DSLs for IaC. Yet, they greatly differ in their purpose, flexibility and other parameters. This chapter aims at identifying the differences, comparing them and finally selecting the most appropriate DSL to be extended to bare-metal.
Since in most cases there is no obvious perfect solution, the selection process needs first gather the most prominent options. Because there are many DSLs, languages need to be ruled out based on their limitations afterwards. As an intermediate result, two or three languages should remain. These can than be compared

on a deeper level, for example whether their internal design allows to easily extend it. Based on the better understanding gained in that step, a meaningful decision can be made.

3.2.1 Amazon CloudFormation

CloudFormation supports both JSON and YAML, is declarative and typed [86]. The typing is done with an additional field "type" for all components. An example type is `AWS::EC2::Instance`, so it has the format of `AWS::ProductIdentifier::ResourceType` [86].
Instead of requiring a commandline-tool (there exists one [87] though), CloudFormation is designed to work by just uploading the file containing the definition - possible sources are s3-buckets, git-repositories or manual uploads. This implies that the orchestrator is run closed-source by Amazon. Therefore CloudFormation is not only a language by Amazon, but also exclusively for Amazon. Additionally, the user has no (direct) influence on the capabilities of the language and the orchestrator.
Nevertheless, AWS holds by far the largest market share of the cloud market [88] and was the first public cloud provider. CloudFormation is therefore one of the earliest DSLs for describing infrastructure. It is widely used [89], and the language itself as well as the tools around it can be assumed to be very mature. There are plugins for most IDEs [90]. While the open source linter doesn't guarantee to be all-seeing and perfect, it at least promises to not fail in case it doesn't understand everything [90]. Under the hood, the linter uses schema validation. Assuming the schema is as mature as the language, it can be reasoned that this guarantees validity of the definition files. The linter also provides detailed information on what exactly is wrong in such a file as well, making it quite error-prone. There is a so-called "AWS CloudFormation Designer", too [91]. It aims at giving the user a GUI to create his infrastructure definition files.
As do most languages for IaC, CloudFormation supports custom functions, too. It does so in both JSON and YAML. Examples are
`{ "Fn::GetAtt" : [ "logicalNameOfResource", "attributeName" ] }` in the former language and `Fn::GetAtt: [ logicalNameOfResource, attributeName ]` or the short-version `!GetAtt logicalNameOfResource.attributeName` in the latter.
Using CloudFormation to describe infrastructure requires a lot of knowledge: Starting from all the products and features AWS has to offer, over different solutions that have (partially) redundant features, up to understanding the AWS jargon. An example for this is "EC2": New users have a hard time understanding that "EC2" is actually a VM and that there is no "EC1" or similar.
Since CloudFormation is limited to AWS, it is incompatible with bare metal. This also means that it can be ruled out for further usage in this thesis. Nevertheless, it is a big player in the league of DSLs, so examining it for reference does definitely make sense (to some extend).

3.2.2 OpenStack Heat

The Heat component from OpenStack is responsible for IaC. It is not a language by itself, but supports actually two languages. One is named Heat Orchestration Template (HOT) and the other is Amazon CloudFormation. The former is strongly influenced by CloudFormation [92]. When the API for CloudFormation is used, the Heat component translates the AWS-specific types to OpenStack compatible ones [93].
HOT is designed very similar to its counterpart from Amazon, too: They have the same type system (with different types though) and the same overall structure [93]. The contextual jargon (f.e. "stack") is also inspired by CloudFormation.
Another similarity is about the required knowledge about products/components. Sticking to the earlier example of creating a VM, new users are required to know that the necessary type is "OS::NOVA::Server". The "NOVA"-part comes from the fact that the compute component of OpenStack is named this way.
There are three ways to communicate with the Heat component; First, the CloudFormation cli-tool. Then an additional commandline tool for HOT [94] and an official library in/for python [95].
OpenStack supports bare metal via a component called "ironic" [96]. It is the closest implementation of what this thesis desires to accomplish [97]. It supports software-defining how new nodes should be provisioned and implements all necessary features - together with other OpenStack components like neutron for networking, glance for OS images, keystone for service discovery and nova for compute node management (f.e. metadata).

3.2.3 HashiCorp Configuration Language and Terraform

One of the most prominent tools is Terraform by HashiCorp [48]. When it was introduced in 2014, it was primarily focused on Amazon Web Services (AWS), but it evolved a lot since then. Nowadays, Terraform supports far over a thousand providers [98]. Of those providers "only" 160 are aimed at IaC [99]. Terraform uses HCL as DSL and is highly plugin-based [100] [101]. In addition to the custom language, JavaScript Object Notation (JSON) is supported as well [102].
Working with Terraform happens with a commandline executable. The binary loads plugins as needed, communicates with the APIs of the necessary providers and gives feedback to the user. In order to be as efficient as possible, Terraform maintains a local state file. The file contains the last obtained state of the infrastructure. Since Terraform assumes it is the only component changing the infrastructure, this approach enables it to detect differences locally. Afterwards, it automatically generates an execution plan on how to eliminate those differences and reach the desired state. The last step is then the execution itself, which is at the same time the only step where communication with external APIs happens.
Cross-plugin dependencies are supported by the tool as well. This makes Terraform extremely versatile and easily extensible.

Since the state file is meant to mirror the current state, manual interactions with the infrastructure as well as with the state file are strongly recommended against [103]. Another reason is the difficulty of recovering from such state disasters [103].

3.2.4 Pulumi

Pulumi is a relatively new technology. Since its first public release in 2017, it came a long way and now advertises as "IaC for any cloud with any language" [104] [105]. Instead of using Yet Another Markup Language (YAML), JSON or HCL, Pulumi is available as library in several programming languages. Available are these libraries for Node.js, JavaScript, TypeScript, Python, Go(lang), and .NET Core (therefore C#, F#, and Visual Basic).
The very different approach Pulumi takes is extremely interesting: It takes the "as code" part of IaC to a new level. On the other side it has no documentation on how to extend it and the current state only supports public clouds and kubernetes [106]. This renders it basically useless for the scope of this thesis.

3.2.5 Open Cloud Computing Interface

[https://www.opentosca.org/documents/Presentation_TOSCA.pdf]

Two non-vendor-specific standards for describing IaC in a formal way have emerged. First, Open Cloud Computing Interface (OCCI) which was published by the Open Grid Forum (OGF) Open Grid Forum in 2011 https://www.ogf.org/documents/GFD.183.pdf. Their organizational member list mirrors their mainly academic purpose [107]. Yet, the website of the OCCI standard reveals that the last contribution happened back in 2016, so this project seems to be either abandoned or at least neglected since then.
OCCI defines a protocol and API for a range of management tasks [108]. Initially designed for Infrastructure-as-a-Service (IaaS), it has since evolved to serve other models like Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) [108]. OCCI is developed by the Open Grid Forum, which is backed by companies such as Dell EMC, NetApp and Oracle [109] [110]. Since the launch of the standard, several open source cloud providers have started to support it, including OpenStack, OpenNebula and CloudStack [111]. The last update of the specification was in 2016 [112]. The IT worlds changes fast, so five years since the last update are a long time.

3.2.6 OASIS TOSCA with Simple-Profile

The second cross-vendor standard that has emerged is called Topology and Orchestration Specification for Cloud Applications (TOSCA). It was first published in

2013 by the Organization for the Advancement of Structured Information Standards (OASIS). The latter is also well-known for widespread standards like Advanced Message Queuing Protocol (AMQP), Message Queuing Telemetry Transport (MQTT), OpenDocument, PKCS#11, Security Assertion Markup Language (SAML), SARIF and VirtIO [113]. Additionally, its members are not only an overwhelming number of academic or governmental institutions but even more so global players like Cisco, Dell, Google, Huawei, HP, IBM, ISO/IEC, the MIT, SAP and VMware [114] [115] [116]. The latest contribution was only one week before the time of writing, so its actively pursued and developed [117] [118].
TOSCA has been used in some proof-of-concept projects [Domain-specific language for infrastructure as code] in 2019, but their results were disappointing: The interfaces between the core standard and the supported providers are described as to be always out of date making even simple operations impossible. The tools of the ecosystem surrounding the standard are said to be non-user-friendly and their learning curves to be flat [119]. Still, TOSCA has a lot of plug-ins for platforms like OpenStack, VMWare, AWS, Google Compute Platform (GCP) and Microsoft Azure (Azure), configuration management tools like ansible, chef, puppet and saltstack or container orchestrators like docker swarm and kubernetes [119], [120]. All those projects conclude that the standard is extremely promising, but the current state makes it impossible to use properly [119].
Since then, TOSCA 2.0 was released, which introduced huge changes like the transition from XML- to the current YAML-declaration.
The standard contains the specificiation of a file archive format called Cloud Service ARchive (CSAR). These archives contain five major parts `https://www.opentosca.org/documents/howto-build-csars.pdf`:

- Type definitions, where properties and interfaces are defined
- A topology template, that describes the overall design and how the types should interact.
- Deployment artifacts, like images and binaries
- Implementation artifacts, like scripts
- Management plans that describe certain actions, f.e. how to instantiate a new VM

In addition to the TOSCA standard itself, OASIS also published an extending standard called “TOSCA Simple Profile” [121]. While large parts of both specifications are redundant, Simple Profile provides the types and ecosystem needed for real-world applications of TOSCA. The reference implementation of the TOSCA orchestrator, OpenTOSCA interprets and executes whatever is necessary of a CSAR definition is also compatible with the TOSCA Simple Profile `https://www.opentosca.org/documents/howto-build-csars.pdf`.

3.2.7 OASIS TOSCA with Cloudify

While both TOSCA implementations / extensions of the TOSCA standard are closely related, their included types and ecosystem are different. Yet, their approach is similar: Both have their orchestrator implemented as backend of a webapplication. Both allow visualization and (partial) graphical editing of the CSAR definitions. Both provider the user with a catalog of example use-cases.
Cloudify is plugin based, which allows it to support different providers on different levels and for different areas [122]. It is very well documented and has many useful integrations like LDAP for authentication and authorization [123].

3.2.8 Ansible

Being one of the best-known IaC tools, Ansibles user base is huge and it can be considered very mature. It does support power cycle management and overall BMC interactions for some hardware providers like Hewlett-Packards remote management software iLO and DELL EMCs counterpart iDRAC [124] [125]. But for "general" bare-metal provisioning it mostly relies on external systems like cobbler [126].

3.2.9 Others

There are really a lot of DSLs around IaC. Many of them were not introduced here. The main reason is that their purpose is different or their approach doesn't fit. As an example for the latter, cobbler is only about hardware provisioning, but it is not designed for managing virtual machines or describing applications that run on them. On the other side, VMware vSphere can do most of that (it is not made for describing applications though), but it is primarily GUI based, and therefore only partially usable for IaC.

3.3 Exclusion based on limitations

The tools and DSLs around IaC can mostly be split up into two fields: On one side is the provisioning, where instantiation is the main purpose. On the other side is configuration management, where instantiation is "assumed" and the goal is to change configurations. Some of the most promiment examples are Terraform, Cloudformation, Heat, and Vagrant for provisioning and Ansible, Chef or Puppet for configuration management [127] [38] [128] [129] [130].
These two categories are named differently in different sources, f.e. using "Infrastructure Definition" as synonym for provisioning and "Configuration Registry" for configuration management. Some software like Ansible can also fill both roles [38].

The field of configuration management is mainly platform-agnostic. Or more specifically, it does not matter whether it is applying to cloud instances, VMs or bare-metal. Most of the tools in this area interact with a preexisting API for their tasks. This could be the API of a cloud provider or SSH access to a VM or bare-metal machine.
In order to bootstrap a whole infrastructure with not preexisting APIs except for the ones available at a hardware level, this thesis focuses on DSLs that are aimed at provisioning. At the same time, it is relatively easy to create instances (of whatever) by sending requests to a cloud provider or a hypervisor, while doing the same with bare-metal not so much: There is no such API - yet.
As a result, all configuration management DSLs are not eligible. Namely, these are Ansible, Chef and Puppet (from the introduced languages at least).
As described earlier, all languages that are imperatively describing infrastructure can be ruled out as well. Funnily enough, this doesn't apply to any of the introduced DSLs for IaC.
As already stated earlier, Amazon CloudFormation is too AWS-specific to be easily extended to bare-metal and is therefore also not an option.
The implementations of OCCI-compatible software are severely outdated, and their interrelated deprecation/maintenance levels are confusing to say the least [131] [132] [133] [134]. That, and because the latest release of the OCCI standard was over five years ago, as well as sources stating OCCI software does not work even with basic examples, together with an accompanying recommendation to use TOSCA instead, OCCI will not be looked at in too much detail as well. For example, the documentation for the OpenStack implementation [131] recommends to visit the corresponding wiki-site, which is even older [135].
Due to its origin, OpenStack Heat definitely has the ability to possibly solve the initially described problems of this thesis. Sadly enough, its documentation leaves much to be desired [136]. Additionally, there are few examples, and to run it even in a proof-of-concept style requires to install and run many components of OpenStack. These requirements make it not only unfeasable for smaller infrastructures, but for ”from-scratch” deployments like the one desired in this thesis as well.
While Pulumi has an extremely interesting approach with using actual programming languages as medium, it is neither easily extensible nor does it aim at provisioning infrastructure itself. Instead it parses the infrastructure code and communicates with the corresponding providers. Based on both of these limitations Pulumi is not a valid potential candidate for the scope of this thesis.
TOSCA has several implementations and corresponding (inofficial) extensions like Cloudify, Alien 4 Cloud or Puccini. It is ouf of this thesis' capabilities to compare all of them. Therefore, only the official extension, named “Simple-Profile” will be included in the comparison.

Table 3.1: *Overview of language candidates that are ruled out*

Language	Reason
Ansible, Chef, Puppet	Ruled out because they are made for configuration management, not provisioning.
CloudFormation	Ruled out because it is too AWS-specific.
OCCI	Ruled out because of old/outdated specificiation and tools.
Heat/HOT	Ruled out because of unfeasable prerequisites.
Pulumi	Ruled out because only public clouds and kubernetes are supported
inofficial TOSCA extensions	Ruled out because too many and closely related to original TOSCA.

3.4 Dimensions of a comparison

Choosing and selecting the best language for a task is hard. Not only is it hard to agree on what is important to compare, nor is it just time-consuming, but it is greatly domain-specific as well. There exist multiple comparisons or -methods for DSLs already, most of which are not infrastructure-specific [137] [85] [46] [138]. They compare based on attributes like (no specific order):

- Primary approach [137]
- Guarantees provided in case of well-formedness [137]
- Reusability of components [137]
- Error proneness and reporting like line number and column offset [137] [85]
- Efficiency: Amount of code for a given case study [137]
- Aspects to learn for a given case study or how hard the mental operations are [137]
- Viscosity: How hard it is to make changes/updates [85]
- Hidden dependencies like requiring agents, a dedicated server or a third-party software [85]
- Visibility: How easy is it to find the responsible snippet in the codebase [85]
- Extensibility: Can the language be adapted to environment changes
- Maturity (documentation, user-base, community): How good are edge-cases documented and how well is the product established
- Ecosystem

The landscape of infrastructure is ever changing - and so are the used tools and protocols. Therefore, “extensibility” is another property this thesis is going to consider.
Also, younger products tend to change a lot at the beginning, while older products have a hard time coping with change. Because of that, another property that is going to be compared is the “maturity”. It also relates to the covered edge-cases which takes into consideration the size of the user-base and the quality and quantity of the documentation.
Very important for the DSL in this thesis is the “ecosystem” surrounding it. This aims at the software that interprets the language, derives actions from it and executes them.
Some of the chosen sources describe more dimensions for their comparisons. While these are useful in general, it was either clear that all languages would perform the same or they are specifically hard to measure (objectively or in a feasable amount of time).
It is important to note that it is out of this thesis’ scope to compare the languages on a deeper level, as for example their abstract syntax (i.e. meta models) or their (Extended) Backus-Naur forms. While the selection process is an important part, the goal of this thesis is not to find the perfect DSL but to find a fitting one and extend it so it can be applied on bare-metal.

3.5 Comparison

Previously, many DSLs were ruled out, so this thesis is going to look at the remaining three languages in more detail. These are HashiCorp Configuration Language (HCL) with the tool Terraform and TOSCA in combination with its “Simple-Profile” extension.

3.5.1 HashiCorp Configuration Language and Terraform

The Configuration Language used by Terraform is a mixture between JSON, YAML and a programming language like Golang. To give a first impression, the JSON in code-snippet 3.1 expresses the very same as the HCL in code-snippet 3.2.

```
"resource": {
  "aws_instance": {
    "example": {
      "instance_type": "t2.micro",
      "ami": "ami-abc123"
    }
  }
}
```

Listing 3.1: *JSON example*

```
resource "aws_instance" "example" {
  instance_type = "t2.micro"
  ami = "ami-abc123"
}
```

Listing 3.2: *HCL example*

The language is able to display the same amount of information in a denser way. This has both positive and negative effects: On one hand, the fewer brackets enable users to focus on the actual content. On the other hand, it is its own language, and developers first need to learn it.
The architecture of Terraform is highly plugin-based and these plugins are often developed by third parties. Terraform accomplishes its integration with those via Remote Procedure Call (RPC). This means, that each plugin is an executable on its own. And when deploying with Terraform, it simply invokes the necessary executables. When invoked, plugins could use a library dedicated for the provider, communicate with APIs or invoke other executables. This approach also allows plugin developers to decide on the programming language of their choice. At the same time, it outsources authentication for each provider to the corresponding plugin. The core Terrafrom executable can detect necessary plugins out of the provided code and downloads them automatically from a repository like the Terraform registry.
Still, the plugin system is cure and blessing at the same time. It enables extraordinary extensibility, yet makes it hard to reuse snippets - what works for one provider most often doesn't work for another that provides basically the very same feature-set. On the other side, being able to share the same provider integration across organizations is a huge bonus that reduces overall duplicate work. The system allows for cross-plugin dependencies, so that really every infrastructure integration can be described. But the quality of the ecosystem depends on the quality of the plugins in most cases. Since Terraforms user-base is huge, most plugins are relatively mature. This means most edge cases are covered, the documentation is great, and most features of a provider are available.
In larger environments, making the correct change to the infrastructure described

with HCL can be hard. Not only does the developer need to know all used technologies, platforms, providers and their specific products and jargon used in the state description, but it has to determine which one relates to the bit it wants to change as well.
Using Terraform comes with three steps: Authoring the infrastructure configuration as code, (automated) planning on how to achieve the desired state and applying/provisioning the desired infrastructure [139]. The authoring part is self-explaining. The planning is done automatically by the Terraform executable [139]. It compares the state described in the code with the current state, and create an execution plan, where things are parallelized as much as possible.
Instead of retrieving the current state via APIs directly from the providers, Terraform maintains a local state(-file). Apart from mapping configuration to real world instance identifiers (like an VM-identifier for example) it is used to significantly improve the performance of the planning phase [140]; The local mirroring of the actual state enables Terraform to work with the state as fast as a local file-accesss instead of requiring several API-requests. This is especially convenient in large environments, where the provisioned state consists of hundreds or thousands of instances [141]. The local caching of the state has the downside that it is required to sync the state of all invocations of Terraform for the same infrastructure [141] in order to prevent race conditions when deploying. If Terraform is integrated in a sole Continuous Integration / Continuous Deployment (CI/CD) pipeline this is not a problem as there is only one instance. For all other cases the documentation of Terraform recommends to use a so-called remote state backend that provides state locking as a main feature [142]. Supported backends range from a shared folder over Terraform Cloud to S3-compatible object storages.
It is strongly advised against manipulating the state manually [140]. Errors are hard to debug, since Terraform assumes that the state is always valid. Apart from manual changes, failures during state-migration (when applying a new state and writing it back to the statefile) can result in almost unrecoverable crashes of Terraform [143].
At the same time it is strongly advised against manual changes in the infrastructure without using Terraform to get there. After such changes, Terraform might be unable to find that component later and assumes it doesn't exist.

3.5.2 OASIS TOSCA with Simple-Profile

As described earlier, in TOSCA infrastructure is modelled via CSAR files. As described in figure 3.1, these archives contain multiple parts. These can be defined in one or multiple YAML files. The main component is the so-called Topology Template. It describes how the different parts of a system interact with each other, gives a holistic overview and defines variable values.

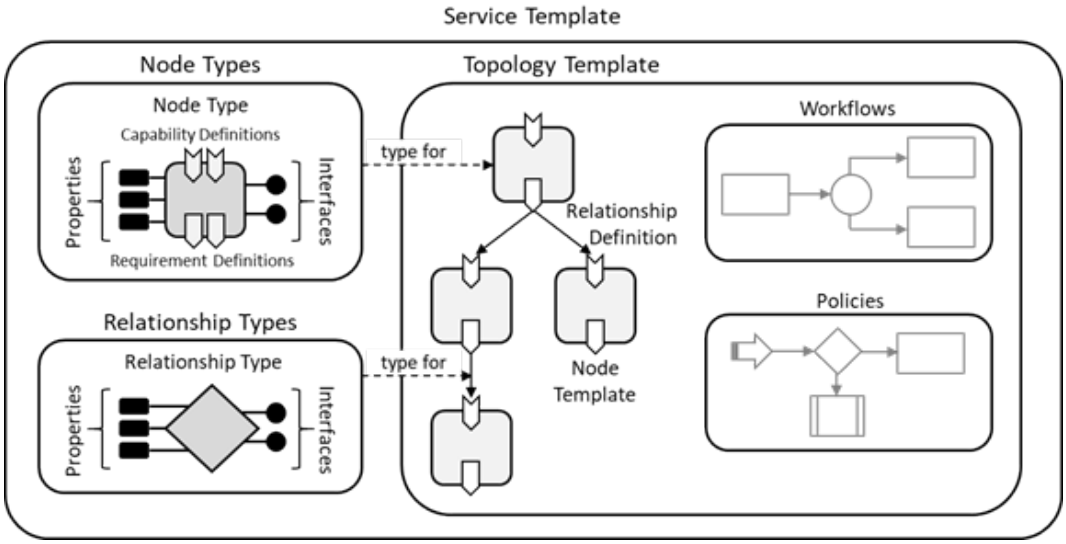

Figure 3.1: *Architecture of TOSCA and the components of CSAR files [117]*

Because the parts (for example a webapplication on a webserver that references a database on another server) can often be reused for other infrastructure parts (or other applications), it makes sense to split a generic description for those parts from the concrete case-specific definition. TOSCA does this with its type definitions. They are optional and not part of the Topology Template, and their sole purpose is to provide a generic template of a component. They can best be desribed as building blocks, whereas the Topology Template is the actual recipe. It is important to not get confused by the fact that TOSCA has three layers of abstraction: The most generic description of a component is the corresponding type. The type defines which properties are allowed, required, what their datatype is and which connections to other components are allowed or required. Derived from this type is a template. As already stated, the template is part of the Topology Template, which describes the actual desired state. The template contains all the information required to instantiate it.
The standard allows for different kinds of types; The most important type is the node type. In TOSCA context, the term “node” does not only refer to machines or servers, but all components that can be defined via software. This includes low-level applications like operating systems, intermediate middleware programs like web- and database-servers, as well as high-level software like the webapplications running on them. The other types have a more specialized purpose. For example artifact types are used to describe constant artifacts like OS-images, scripts and other external (as in non-TOSCA) files. The type categories are data-, capability-, interface-, relationship-, group- and policy-types. For most types there are predefined/default ones, as the basic YAML types like string, integer, etc. for datatypes. Custom ones can be added though, as well as composite datatypes. The other types are either self-explaining or not necessary to understand for this thesis. Except for the capability and requirement system, which is worth additional explanation. In order to reflect where relationships are allowed (for example a webapplication cannot run on a database server), the standard has the aforementioned

system. Corresponding capabilities interlock with requirements like puzzle pieces. An example: Assuming a node has a requirement “container runtime”, it can only be assigned to an underlying node that offers a capability with the same name. To satisfy this constraint, TOSCA orchestrators know they need a node that offers that capability. If there isn’t one, they will try to create one. If the container runtime itself has other requirements like “compute”, the orchestrator will resolve those as well. Figure 3.3 displays how types and templates correspond and how requirements and capabilities are mapped.

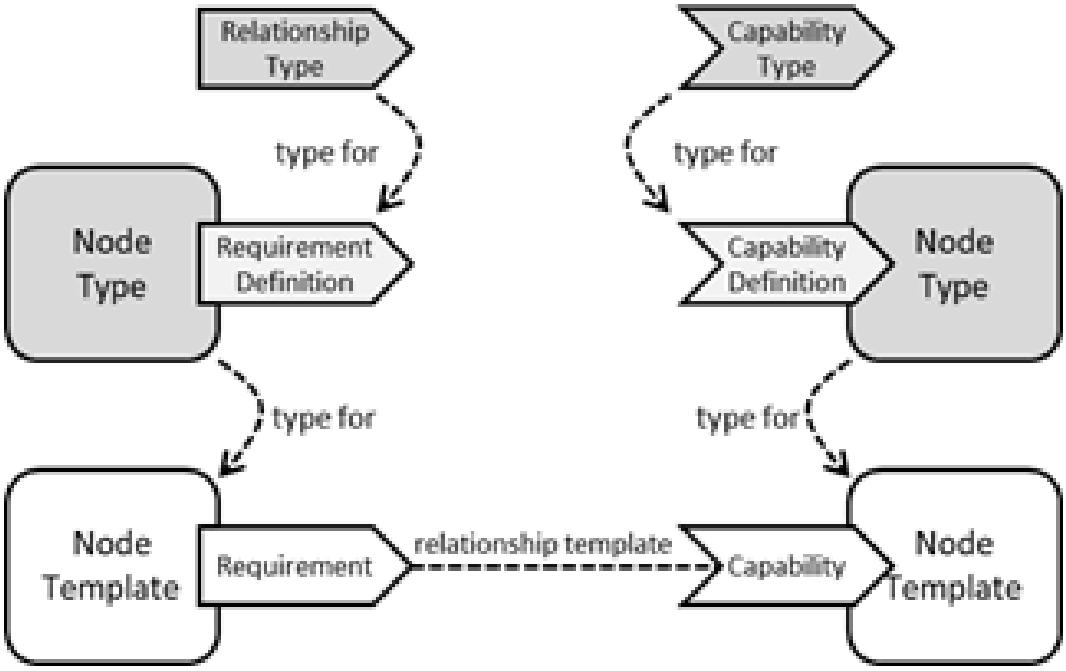

Figure 3.2: *Derivation and relationships between TOSCA elements [117]*

The last part contained in CSAR files are the management plans. They describe the lifecycle of nodes and how to achieve them. Examples are instantiation, configuration and deallocating/uninstallation/deletion (depending on their type) of nodes [117].
The standard supports imports with or without namespacing. One common import is the Simple-Profile extension. It adds a basic set of predefined types to the standard, for example the node-types Compute, Webserver, and Webapplication with necessary (default) properties, requirements and capabilities.
Another important feature of the TOSCA standard is “substitution”. It allows users to outsource the definition of a specific node template to a completely different CSAR package. This allows splitting and distribution of concerns and strongly increases reusability of components. The architecture of the CSAR files and the ability of inclusions and substitutions make it also relatively easy to find the corresponding code for a certain component. Figure **??** visualizes this substitution approach.

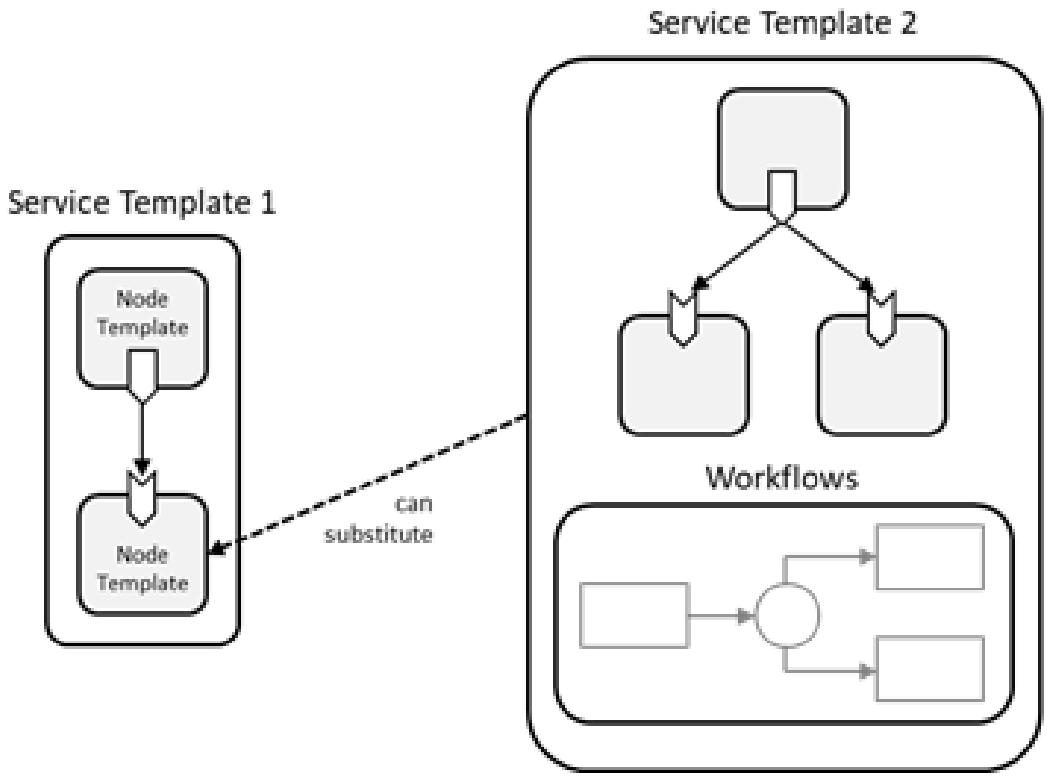

Figure 3.3: *Substitution of node templates with external topologies [117]*

While TOSCA does not seem to be very widespread, its origin from OASIS and the many and huge companies backing it and involved in its development make it a very promising standard. The current state of the specification is unfinished, and it is hard to get started in other ways than reading the specification.
The two most commonly used implementations of TOSCA are OpenTOSCA and Cloudify (the latter with its own TOSCA-extension). All those preexisting orchestrators have the same approach: They have a web-based frontend, where deployments are managed. In some cases like Cloudify, an additional commandline tool exists, which communicates with the web-based orchestrator via an API. The orchestrator is therefore needed to run at all times, and needs to have access to the whole infrastructure. The web-based design allows for easy integration with authentication services like LDAP.

3.5.3 Selection

As can be seen by both the amount of initial candidates and the length of the selection process, making a decision for a DSL is hard. In this case, it "only" influences the further course of the thesis, but organizations have the same struggle (based on the amount of comparisons and recommendations that circulate the internet). This is due to the nature of these languages: As soon as one is used, it is not feasable to move to another soon. The organization is locked into it. What cross-platform tools and languages like Terraform accomplish is abstracting the vendor

lock-in away by language lock-in. Before the rise of IaC, organizations had to go through a similar selection process for their providers. With IaC, organizations now have to select a language that supports all potential providers, has a huge and active user- and developerbase, so all development on it distributes across many organizations.
As of 2021, Terraform is definitely the state of the art: It supports more providers than any organization should want to use simultaniously, has the largest market-share and is considered by most providers as a key player [144]. Therefore, its ecosystem grows and matures exponentially. It has achieved economies of scale and almost a monopoly on cross-provider IaC.
TOSCA on the other hand is backed by many important companies and is even more customizable than Terraform. Instead of the plugin-system, it is based on importable components. While Terraform supports multiple providers in the same infrastructure, moving between providers for the same feature (for example switching the cloud provider) requires manual work. With TOSCAs namespacing and substitution system, it is possible to be completely provider-agnostic. At the same time, it does not force (or at least strongly recommend) a new language on its users, as does Terraform with HCL. Additionally, the preexisting types of TOSCAs Simple-Profile extension already include types that are necessary for bare metal, for example the image file type. Another point goes to TOSCA because its extensible design makes it possible to include foreign DSLs for their dedicated tasks. An example is that implementations can be written in bash- or python-scripts
Because TOSCA has a more generic approach and already somewhat aims at working with bare metal, this thesis is going to work with this DSL.

3.6 Reference infrastructure

- Are VMs dead? / will containers replace them completely? (/ the case for bare-metal) - isolation level - comparison of bare-metal approach vs vSphere and/or OpenStack approach - constraints like - Workload comparison; are there workloads which cannot run in containers and require VMs? - minimum machine size defines minimum cluster size and therefore introduces unused resources (when going for temporary k8s-clusters for devs) - -> VMs make sense! What about their overhead? They need ”zone/node affinity” as well - kubevirt? - common components: - public or not (dns / routing) - load-balancer / ha - persistent or not / storage - web-service / api -> should mirror most applications and uses other components - db-api - web-api - REST(ful)-API / CRUD (create, read, update, delete or in HTML: put, get, put, delete, or combine with post) - ACID - identity / email ? - function-as-a-service / serverless -> special case - trend: - https://en.wikipedia.org/wiki/Resource-oriented_architecture, https://en.wikipedia.org/wiki/Resource-oriented_computing, https://en.wikipedia.org/wiki/Service-oriented_architecture, https://en.wikipedia.org/wiki/Web-oriented_architecture - include example in reference architecture? - open data protocol https://en.wikipedia.org/wiki/Open_Data_Protocol - https://en.wikipedia.org/wiki/RSDL - https://en.wikipedia.org/wiki/OpenAPI_Specification (formerly swagger) - - hw-security - limit available OS

images; optimize those for own hw -> less generic drivers, no overall driver-issues, less to support - three installation flavors: - install with pxe - install with attached iso (via ipmi or hypervisor) - preinstalled virtualdisk (only for vms) -> azure - ibm supports only attached iso: `https://cloud.ibm.com/docs/bare-metal?topic=bare-metal-bm-mount-iso` - firmware - some hw supports firmware flashing from os level which can result is hardware damage (increasing voltage etc) - either on provision or deprovision task update all firmwares to latest official firmware versions (no matter what was installed before - even if it seems to be that already) - on deprovisioning makes more sense, it saves time when provisioning new nodes. - upgrades can then happen globally (for all "unused" nodes) and used nodes can be migrated by users (or not...) - allow to select which firmware version to have flashed - latest is default - fix them to current latest version after latest was used - `https://docs.microsoft.com/en-us/azure/baremetal-infrastructure/concepts-baremetal-infrastructure-overview` - ? bare metal is ISO 27001, ISO 27017, SOC1 SOC2 compliant - RHEL and SLES only - ECC vs EDAC (Error Detection And Correction) module; ECC is in hardware, EDAC in software, when both enabled, they can conflicts, with unplanned shutdowns of a server. - managed bare metal; up to OS is managed, then the customer is responsible

-

4 Design and Implementation

4.1 Requirements

The orchestrator should understand TOSCA, extensions from it and should be able to provision bare-metal machines. Not only is it possible to split this into subtasks, but it makes sense as well. By doing so, the modules can be developed and updated one after another, in parallel if necessary and at a later point even interchanged with other implementations. In order to slice the application into reasonable packages, their domains should not overlap, and their external interface as small as possible.
To achieve that goal, the application workflow needs to be analyzed:
At the beginning, the CSAR files, its content and the YAML structure of the content needs to be parsed and validated. This includes handing down properties of both type-from-type and template-from-type derivations, as well as enabling (namespaced) imports. Then, the orchestrator must be able to wake machines with WOL. After a machine is powered on, it attempts to boot over network. Therefore, the orchestrator has to manipulate an external or internal (as in “integrated in the orchestration software”) DHCP server. In order to provide the orchestrator with the necessary information about the machine, it makes sense to use a live-OS that does not need to be installed, but can be booted directly. Optimally, it should be relatively small, since its transferred via network, boot fast, and somehow provide the orchestrator with information about the underlying hardware like its RAM-size. The last step is then running commands on the machine like installing a package or copying files to it. Optimally, the user should be informed on what is currently going on during the whole process.
Figure 4.1 shows the whole workflow in an interaction overview diagram.

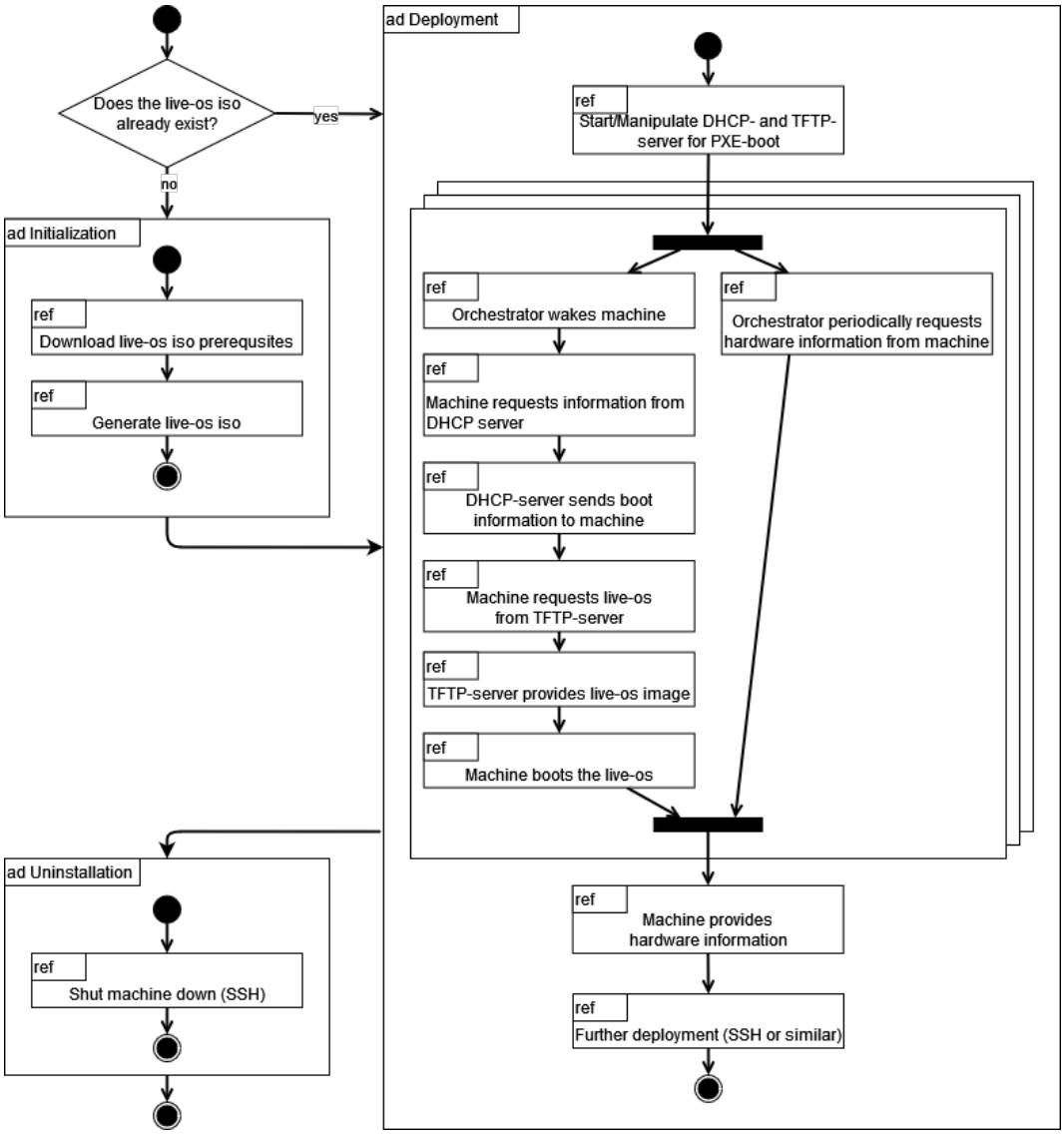

Figure 4.1: *UML interaction overview diagram describing the application workflow*

4.2 Architecture

To diminish the limitations of a web-based approach for the orchestrator like the hogging of resources even during idle times, the goal is to have one or many libraries, and a commandline-based wrapper around it.
As described above, the tasks of the orchestrator can (and should) be split into different steps. For example the wake-on-lan part and the DHCP server have nothing in common except both being invoked from the orchestrator whenever they are needed.
The executable has at least three subcommands; One for initialization, where the live-OS image is generated and the DHCP server is prepared. And a second where the actual deployment happens. Last but not least, an uninstallation subcommand

is needed to reverse the deployment.
The following chapters describe how the domains within those subcommands are sliced in order to have different modules for the different domains.

4.3 Packages

Most of the packages described in the following chapters strongly relate to the steps described in figure 4.1. They are described in the order they are invoked during both the initialization and deployment subcommands. A summary of interactions and dependencies is shown in the package diagram in figure 4.2 below. The most important dependencies are marked with "«use»", whereas simple type imports are marked with "«import»" and have a dashed line.

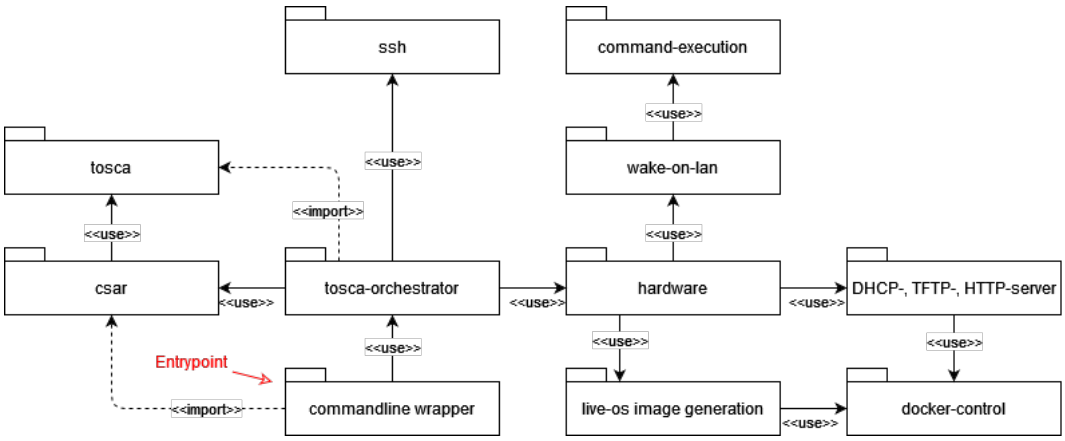

Figure 4.2: *UML package diagram describing how the packages are related to each other*

As can be seen in the diagram, the commandline-wrapper communicates only with one package: The orchestrator, which does all of the actual work. For non-bare-metal applications, the orchestrator only uses the CSAR, the SSH, and the command-execution package directly. They are sufficient to be compliant with the current standard specification. When bare-metal machines should be provisioned, the orchestrator also communicates with the hardware package. It manages the generation of the live-OS image, the interactions with the DHCP-, TFTP-, and HTTP-servers, as well as the wake-process of the bare-metal machines.
When invoked with the initialization subcommand, the application assumes hardware will be provisioned and generates the necessary artifacts.
Deployment of applications starts with parsing the CSAR contents. While the CSAR package handles file-I/O, the TOSCA package is the actual parser.

4.3.1 TOSCA

In order to be able to implement a library for the TOSCA standard, it is necessary to work through both the original specification and the Simple-Profile extension counterpart, since both are meant to work together. Important information is sometimes distributed over both specifications, so to fully understand it, this is necessary. Additionally, preexisting libraries for the chosen programming language might exist. In case of this thesis' reference implementation Golang is the language of choice. Sadly, most of the larger libraries are based on one another or unfinished, and the most complete of them are severly outdated [145].
In order to support the latest version of the specification, and to be able to easily extend it if necessary, a complete reimplementation of the TOSCA library is created. It is strongly influenced by the largest but outdated one, but is more complete. Only with the already described line-by-line read-through of both specifications it is possible to gather enough information about the standard and all its features.
With the new library, it is possible to parse any (valid) Types, Templates, and Service Topologies. In addition to parsing them and creating Go-native structures out of them, basic functions like "get_input" and "get_attribute" or validation of elements, it is limited to one implementation type, Bash, while the specification states that Python should be supported as well. This is due to the proof-of-concept nature of this thesis. Sadly, there are several occasions in the specifications, where it is unclear or simply incomplete. These cases are mostly about edge cases and were not required for the proof-of-concept this thesis tries to achieve. Therefore, they were left out of the implementation. Other cases, such as inconsistencies or required assumptions will be covered in the chapter "Analysis".
Because the basic TOSCA specification describes how to work with extensions like the TOSCA Simple Profile, it is possible to implement such imports, references, and relations as well. This means, the implemented TOSCA library (called package in Golang context) is fully compatible with the TOSCA Simple Profile and all other extensions.
The package is also meant to solve the type derivation, resolve imports, namespacing and validate all of them.

4.3.2 CSAR

After the TOSCA package is able to parse the contents of files, the file-I/O - desribed in the last chapter of the specification - is implemented as a seperate package. It contains information about how to pack multiple artifacts like OS-images, definition-, or other required files together into one CSAR file. The file is basically a standard zip-archive, but the content needs to follow a certain schema. For example there are three places where required metadata like version and name can be placed. If they are not found there, the whole file is invalid.
The reason behind this seperation are the still very different domains: The TOSCA package parses file contents and provides Go-native types, while the CSAR package

is more about accessing files, checking for their existance and making it possible for the TOSCA package to parse its content. Should another project work with file-contents in a database for example, they would not require the file handling and can import only the TOSCA package.
This package depends on the earlier described TOSCA one, as it has a function that takes a file-/folderpath as input and returns the fully parsed TOSCA topology with fully derived templates. "Fully derived" means that all imports are made and template properties are extended with values from their original type wherever necessary.

4.3.3 Command-execution

In order to run Bash commands from the application itself and retrieve the outputs, it makes sense to build a complete package around command execution. It can later be extended with a corresponding implementation for Python, so the application is fully compatible to the TOSCA specification.
The package provides to public methods. One for running direct commands, and another for running a script that resides at a provided location.
Since TOSCA only introduces Bash and Python, this package requires the underlying system to be compatible with both script languages. Windows for example does not support Bash by default.

4.3.4 Docker control

It is clear that some kind of DHCP- and TFTP-server is required. They could be external applications, but it should not be a hard requirement. In cases where no external DHCP- and TFTP-servers exist, it must be possible to set them up in an easily repeatable way. The same applies to the generation of the live-OS image. As in most such cases, this can be solved with (docker) containers. The goal of this thesis is to bring all required bits together, so the application needs to create the docker images, start the containers (with parameters like volumes and forwarded ports), as well as stop and remove the containers when they are not needed anymore (for example when the provisioning is finished).
The official docker binary (for Linux) is created in Golang as well, and the software is open source. Docker even provides an SDK for other developers to integrate communication with the docker engine into their applications. Sadly, the documentation is sparse and the few examples shown along the SDK are often not enough to get even seemingly easy things like container stopping to work. For this particular case it is necessary to add the container stopping and removal twice: Once, when the application terminates successfully, as the container fulfilled its job and isn't needed any more. And a second time, when the application terminates due to an error somewhere else, and the default termination does not happen. Even "deferring" the container termination does not work. Only when the SIGTERM interrupt of

the wrapper application is “manually” listended for and a function is implemented to remove the container in such a case, the removal is successfull in all cases.
Another obstacle is the retrieval of live logs during the container livecycle and embed the retrieved output in the logs of the wrapping application. This can be solved by creating a buffered streamreader, which is periodically polled and checked for contained linebreaks.
As the application is now able to handle docker containers to provide repeatable setup of the DHCP- and TFTP-server, the next step is to implement a repeatable way of a live-OS image generation.

4.3.5 Live-OS image generation

The huge amount of time needed to install a complete OS just for retrieving information about the hardware has the potential to be a huge showstopper. In order to reduce the time needed for information retrieval, a live operating system is used. These operating systems are not necessarily very different from normal ones, but they can most often run from read-only mediums like optical disks. Examples are all OS installers (that can be burned to CD-ROM or similar) - this means there already are many live systems out there.
Since the live-OS is provided via network, a virtual version of such a system is required. One such file-format for disk images is defined by the ISO 9660 standard [146]. The extension of these files is either `.iso` or `.udf`. Due to the name of both the standard and the extension, these files are often simply called iso-files.
In order to be able to create a iso-file that provides information about the underlying hardware from scratch, only one requirement should exists: A working internet connection. Optimally, a generic preexisting live-image is downloaded and then modified to serve the special use case of publishing the desired information. Since tiny Linux images should be relatively common in times of Internet-of-Things and Raspberry Pis, this should be an easy task. The experience tells a different story, though.
The first linux distribution tested during the implementation phase was “Minimal Linux Live”. It is described as an educational Linux, so its configuration is well-documented [147]. A first test with the original downloaded iso file seemed promising: The filesize is less than 50 MB, the OS boots in under 30 seconds and the due to its educational background, it is easily extensible. When the system is fully up and running, the keyboard (mouse is not supported) input did not work under the tested Hyper-V hypervisor. As this is not exactly necessary, the image generation was extended in such a way, that a webserver is started when the system boots. But with or without this extension, the generated image is not bootable on said hypervisor. Tested were both BIOS and UEFI firmwares - to no avail.
A second promising OS is Alpine Linux. It is a full-blown operating system with focus on small footprint and a fast boot process. Due to its focus, it is very widespread in the container world, and its documentation is extremely good. There is even a dedicated wiki-page for custom iso-images [148]. Since there are additional applications necessary to be installed in order to generate the image, and these applica-

tions are installed with the Alpine Linux package management tool “apk”, the first test was run in an Alpine-docker-container. Later, the same things were tested with an Alpine-VM, with the same result. The generated iso-image is four times the size of the original downloadable alpine iso. Not only is there a huge size difference, but when trying to boot it, a kernel file is not found. As described earlier, the overall documentation is quite good, and there are several predefined build-profiles for custom images. While the size is different for each one, the kernel file issue reoccurs in all cases. Even when inserting the missing file from the original Alpine iso, the generated iso remains unbootable.
Another Linux distribution with good documentation and widespread use is Ubuntu. In the Ubuntu app store, there exists a complete graphical application dedicated to image generation. It works perfectly, and the concept with an autostarted webserver that serves information about the underlying hardware can be considered proven. But since the size of the iso file exceeds one GB, and the boot time is close to one minute, another well-known operating system was tried.
Due to slower release cycles, Debian is considered more stable than Ubuntu. At the same time it has not as many features (preinstalled). Therefore its initial size is significantly lower and its non-teaked performance is generally better. While for custom Debian images there is no graphical tool, the Ubuntu one shows the general approach. The steps can be summarized as follows:

1. Unpack the original iso file.
2. Unpack the contained filesystem-image (squashfs).
3. Simulate a running operating system on the unpacked contents with `chroot`.
4. Make all necessary changes, like installing applications, enable services and adding files. This can include boot-menu adjustments like reducing/removing the timeout.
5. Repack the filesystem-image.
6. Repack the image with the changed filesystem-image.

The resulting image is smaller than one GB, boots in under ten seconds, is as extensible as the Ubuntu image, and due to its textual nature completey scriptable. Especially the last part makes it the perfect fit for putting it in a container. It is therefore the selected way to generate a live-OS image.
The customizations include a webserver that autostarts during boot, and a simple service that also starts automatically during the boot-process. The service runs some commands and stores the output into files that are then served by the webserver. Due to the proof-of-concept nature, only information about the CPU and RAM is served.

4.3.6 DHCP-, TFTP-, HTTP-server

docker container, ports, variable config vs fixed config
why not external ones
why http server, too (A observant reader might have noticed...)

4.3.7 Wake-on-lan

actual wol vs simulated wol for hypervisors

4.3.8 Hardware inspection

periodic checking for new ip, try to reach machine, don't fail if unreachable.

4.3.9 SSH

key-generation, variable key path, key placement on servers / integration in live-os
actual command execution and feedback returning
Why only manipulate live-os, instead of deploying complete new os

5 Analysis

what were the goals, what are the current capabilities
how stable is it, what can it do in real world settings

6 Conclusion

new insights, breakthroughs and limitations of current approach, recommendation for future research
What was done, emphazis on own work:

- looked at several DSLs, their requirements and compared X ones in more detail
- added additional elements to the comparison
- improvement recommendations for TOSCA standard
- implemented new approach of orchestrator and go-library for newest tosca standard
- implemented flexible on-demand bare-metal provisioner
- recommendations for tosca standard: merge properties and attributes, provide more examples, describe a reference orchestrator

7 Outlook

- compare metamodels, find similarities, could lead to common ground
- vendor bioses should support http by default or embed ipxe for network boot. Maybe even allow flashing the network-boot system (remotely?). VMware does support this already for its VMs: https://ipxe.org/howto/vmware
- ipmi like interface for provisioning, f.e. provide (remote) kernel path and parameters (addition to ”local boot”, ”net boot” selection)
- common standard for bmc/ipmi features.
- making all bios settings available over an scriptable interface - bmc is not (universal) enough.
- tosca standard improvements from notes. Example: Two types of script execution: one on the orchestrator and one on the nodes.
- improvements to this orchestrator: deploy complete os, instead of only manipulating live-os
- add integrations with external dhcp- tftp- http-servers.
- architecture improvements: plugin-system, so integrations (dhcp, tftp, http) are easier. Like terraform.

Table of figures

Bibliography

[1] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems - Concepts and Designs, fifth Edition*. Addison-Wesley, 2012 (cit. on p. 8).

[2] Mark Baker. *Cluster Computing White Paper*. University of Portsmouth, 2001 (cit. on p. 8).

[3] Rob Long and Maciek Różacki. *Bayer Crop Science seeds the future with 15000-node GKE clusters*. URL: `https://cloud.google.com/blog/products/containers-kubernetes/google-kubernetes-engine-clusters-can-have-up-to-15000-nodes` (visited on 10/13/2021) (cit. on p. 8).

[4] Till Hoffmann. *Schemavalidation of YAML-files in the Context of Kubernetes & Helm*. Technische Universität Ulm, 2018 (cit. on p. 8).

[5] Martin Fowler. *Martin Fowler: Snowflake Servers*. URL: `https://dzone.com/articles/martin-fowler-snowflake` (visited on 10/13/2021) (cit. on p. 10).

[6] Randy Bias. *The History of Pets vs Cattle and How to Use the Analogy Properly*. URL: `http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/` (visited on 10/13/2021) (cit. on p. 10).

[7] KVM-Switch Versand GmbH. *64-Port KVM-Switches*. URL: `https://kvm-switch.de/en/category-335/from-16-Port-KVM-Switches/64-Port-KVM-Switches/` (visited on 10/13/2021) (cit. on p. 10).

[8] Seth Kenlon. *What is a golden image?* URL: `https://opensource.com/article/19/7/what-golden-image` (visited on 10/13/2021) (cit. on p. 11).

[9] Canonical. *Releases*. URL: `https://github.com/canonical/cloud-init/releases?after=ubuntu-0.3.1` (visited on 10/13/2021) (cit. on p. 11).

[10] Networx Security e.V. *Preboot Execution Environment*. URL: `https://www.networxsecurity.org/de/mitgliederbereich/glossary/n/network-bootstrap-program.html` (visited on 10/13/2021) (cit. on p. 11).

[11] Rackspace Technology Inc. *Understanding Bare Metal service*. URL: `https://docs.openstack.org/ironic/latest/user/architecture.html` (visited on 10/13/2021) (cit. on p. 11).

[12] Shannon Meier. *IBM Systems Virtualization: Servers, Storage, and Software*. Redpaper, 2008 (cit. on p. 12).

[13] Jack Loftus. *Xen virtualization quickly becoming open source 'killer app'*. TechTarget, 2005 (cit. on p. 12).

[14] Stephen J. Bigelow and Alexander S. Gillis. *What is server virtualization? The ultimate guide*. URL: `https://searchservervirtualization.techtarget.com/definition/server-virtualization` (visited on 10/13/2021) (cit. on p. 13).

[15] Kimberly Mlitz. *Current and planned adoption of virtualization technologies by businesses in North America and Europe in 2019, by type*. URL: https://www.statista.com/statistics/1139931/adoption-virtualization-technologies-north-america-europe/ (visited on 10/13/2021) (cit. on p. 13).

[16] VMware Inc. *Accelerate IT. Innovate with Your Cloud.* VMware, Inc, 2012 (cit. on p. 13).

[17] Spiceworks. *The 2020 State of Virtualization Technology*. URL: https://www.spiceworks.com/marketing/reports/state-of-virtualization/ (visited on 10/13/2021) (cit. on p. 13).

[18] Cloudflare Inc. *What is the cloud? | Cloud definition*. URL: https://www.cloudflare.com/learning/cloud/what-is-the-cloud/ (visited on 10/13/2021) (cit. on p. 13).

[19] Cloudflare Inc. *What is a private cloud? | Private cloud vs. public cloud*. URL: https://www.cloudflare.com/learning/cloud/what-is-a-private-cloud/ (visited on 10/13/2021) (cit. on p. 13).

[20] Microsoft Corporation. *What is a private cloud?* URL: https://azure.microsoft.com/en-us/overview/what-is-a-private-cloud/ (visited on 10/13/2021) (cit. on p. 13).

[21] Katie Costello and Meghan Rimol. *Gartner Forecasts Worldwide Public Cloud End-User Spending to Grow 23 % in 2021*. URL: https://www.gartner.com/en/newsroom/press-releases/2021-04-21-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-grow-23-percent-in-2021 (visited on 10/13/2021) (cit. on p. 13).

[22] Miguel Barrientos and Claudia Soria. *Market capitalization of listed domestic companies (current US$) - Country Ranking*. URL: https://www.indexmundi.com/facts/indicators/CM.MKT.LCAP.CD/rankings (visited on 10/13/2021) (cit. on p. 13).

[23] CB Information Services Inc. *Here's Why Amazon Is No Shoo-In To Win The $206B Global Cloud Market*. URL: https://www.cbinsights.com/research/amazon-google-microsoft-multi-cloud-strategies/#history (visited on 10/13/2021) (cit. on p. 13).

[24] Jared Wray. *Where's The Rub: Cloud Computing's Hidden Costs*. URL: https://www.forbes.com/sites/centurylink/2014/02/27/wheres-the-rub-cloud-computings-hidden-costs/ (visited on 10/13/2021) (cit. on p. 13).

[25] Jonathan Corbet. *Process containers*. URL: https://lwn.net/Articles/236038/ (visited on 10/13/2021) (cit. on p. 14).

[26] Jonathan Corbet. *Notes from a container*. URL: https://lwn.net/Articles/256389/ (visited on 10/13/2021) (cit. on p. 14).

[27] Julien Barbier. *It's Here: Docker 1.0*. URL: https://web.archive.org/web/20150306220543/https://blog.docker.com/2014/06/its-here-docker-1-0/ (visited on 10/13/2021) (cit. on p. 14).

[28] Camino Cielo Cloud Services. *VPS Provisioning Times*. URL: https://www.vpsbenchmarks.com/labs/provisioning_times (visited on 10/14/2021) (cit. on p. 14).

[29] CloudPassage Inc. *Why containers are the future of cloud computing*. URL: https://www.cloudpassage.com/articles/containers-future-cloud-computing/ (visited on 10/14/2021) (cit. on p. 14).

[30] Constance Drugeot. *Is serverless the future?* URL: https://www.devopsonline.co.uk/is-serverless-the-future/ (visited on 10/14/2021) (cit. on p. 14).

[31] Buchen. *Why Is Serverless the Future of Cloud Computing?* URL: https://www.alibabacloud.com/blog/why-is-serverless-the-future-of-cloud-computing_597191 (visited on 10/14/2021) (cit. on p. 14).

[32] Paweł Zubkiewicz. *Why serverless is the future of software and apps*. URL: https://ttpsc.com/en/blog/why-serverless-is-the-future-of-software-and-apps/ (visited on 10/14/2021) (cit. on p. 14).

[33] Bill Karagounis. *Introducing the Microsoft Azure Modular Datacenter*. URL: https://azure.microsoft.com/en-us/blog/introducing-the-microsoft-azure-modular-datacenter/ (visited on 10/14/2021) (cit. on p. 14).

[34] William H. Whitted and Gerald Aigner. *Modular data center*. URL: https://patents.google.com/patent/US7278273B1/en (visited on 10/14/2021) (cit. on p. 14).

[35] Ayrat Khayretdinov. *The beginner's guide to the CNCF landscape*. URL: https://www.cncf.io/blog/2018/11/05/beginners-guide-cncf-landscape/ (visited on 10/14/2021) (cit. on p. 14).

[36] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. *How software engineers use documentation: the state of the practice*. IEEE, 2003 (cit. on p. 15).

[37] C. J. Satish and M. Anand. *Software Documentation Management Issues and Practices: a Survey*. Redpaper, 2016 (cit. on p. 15).

[38] Kief Morris. *Infrastructure as Code, second Edition*. O'Reilly Media, Inc., 2008 (cit. on pp. 15, 16, 20, 25).

[39] Ranjit Bawa and Rick Clark. *Software-defined everything - Breaking virtualisation's final frontier*. Deloitte University Press, 2008 (cit. on p. 15).

[40] M. N. O. Sadiku1, S. R. Nelatury, and S.M. Musa1. *Software Defined Everything*. Journal of Scientific and Engineering Research, 2017 (cit. on p. 15).

[41] Paul Venezia. *Review: Puppet vs. Chef vs. Ansible vs. Salt*. URL: https://www.infoworld.com/article/2609482/data-center-review-puppet-vs-chef-vs-ansible-vs-salt.html (visited on 10/14/2021) (cit. on p. 15).

[42] Rob Hirschfeld. *Can Infrastructure as Code apply to Bare Metal?* RackN, 2021 (cit. on pp. 15, 16, 18).

[43] Canonical Ltd. *cloud-init Documentation*. URL: https://cloudinit.readthedocs.io/en/latest/ (visited on 10/14/2021) (cit. on p. 16).

[44] Juan Cadavid, David Lopez, Jesús Hincapié, and Juan Qintero. *A Domain Specific Language to Generate Web Applications*. Researchgate, 2009 (cit. on p. 16).

[45] Thorsten Berger. *Domain-Specific Languages (DSLs) motivation, concepts, examples*. University of Gothenburg, 2018 (cit. on pp. 16, 17).

[46] Valeriya Shvetcova, Oleg Borisenko, and MAxim Polischuk. *Domain-specific language for infrastructure as code*. IEEE, 2019 (cit. on pp. 16, 27).

[47] Melvin E. Conway. *Conway's Law*. URL: http://www.melconway.com/Home/Conways_Law.html (visited on 10/14/2021) (cit. on p. 17).

[48] Google LLC. *Google Trends Infrastructure as Code*. URL: https://trends.google.com/trends/explore?date=today%205-y&q=%2Fg%2F11c3w4k9rx (visited on 10/14/2021) (cit. on pp. 18, 22).

[49] Sheila Manek. *IDC Expects 2021 to Be the Year of Multi-Cloud as Global COVID-19 Pandemic Reaffirms Critical Need for Business Agility*. URL: https://www.idc.com/getdoc.jsp?containerId=prMETA46165020 (visited on 10/14/2021) (cit. on p. 18).

[50] Eric Jhonsa. *Public Clouds Are Bright Spot as IT Outlays Slow Due to Virus*. URL: https://www.thestreet.com/investing/public-clouds-are-bright-spot-as-information-technology-spending-eases (visited on 10/14/2021) (cit. on p. 18).

[51] Ken Downie, John Cobb, Avatar Chris Puckett, and Dan Sisson. *Quickstart: Create an Azure Stack HCI cluster and register it with Azure*. URL: https://docs.microsoft.com/en-us/azure-stack/hci/deploy/deployment-quickstart (visited on 10/14/2021) (cit. on p. 18).

[52] Amazon Web Services Inc. *AWS Outposts*. URL: https://aws.amazon.com/outposts/ (visited on 10/14/2021) (cit. on p. 18).

[53] Amazon Web Services Inc. *AWS Outposts FAQs*. URL: https://aws.amazon.com/outposts/faqs/ (visited on 10/14/2021) (cit. on p. 18).

[54] Google LLC. *GKE on-prem overview*. URL: https://cloud.google.com/anthos/clusters/docs/on-prem/1.3/overview (visited on 10/14/2021) (cit. on p. 19).

[55] Martin Yip. *Farewell, vCenter Server for Windows*. URL: https://blogs.vmware.com/vsphere/2017/08/farewell-vcenter-server-windows.html (visited on 10/14/2021) (cit. on p. 19).

[56] VMware Inc. *Overview of the vSphere Installation and Setup Process*. URL: https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.esxi.install.doc/GUID-B64AA6D3-40A1-4E3E-B03C-94AD2E95C9F5.html (visited on 10/14/2021) (cit. on p. 19).

[57] VMware Inc. *vCenter Server Deployment Models*. URL: https://docs.vmware.com/en/VMware-vSphere/6.0/com.vmware.vsphere.install.doc/GUID-ACCD2814-0F0A-4786-96C0-8C9BB57A4616.html (visited on 10/14/2021) (cit. on p. 19).

[58] VMware Inc. *Installing ESXi*. URL: https://docs.vmware.com/en/VMware-vSphere/6.0/com.vmware.vsphere.install.doc/GUID-D0A72192-ED00-4A5D-970F-E44B1ED586C7.html (visited on 10/14/2021) (cit. on p. 19).

[59] VMware Inc. *Introduction to Auto Deploy*. URL: https://docs.vmware.com/en/VMware-vSphere/6.0/com.vmware.vsphere.install.doc/GUID-9A827220-177E-40DE-99A0-E1EB62A49408.html (visited on 10/14/2021) (cit. on p. 19).

[60] VMware Inc. *Auto Deploy Boot Process*. URL: https://docs.vmware.com/en/VMware-vSphere/6.0/com.vmware.vsphere.install.doc/GUID-8C221180-8B56-4E07-88BE-789B25BA372A.html (visited on 10/14/2021) (cit. on p. 19).

[61] VMware Inc. *Provisioning ESXi Hosts by Using vSphere Auto Deploy*. URL: https://docs.vmware.com/en/VMware-vSphere/6.0/com.vmware.vsphere.install.doc/GUID-71F8AE6C-FF4A-419B-93B7-1D318D4CB771.html (visited on 10/14/2021) (cit. on p. 19).

[62] VMware Inc. *Install the TFTP Server*. URL: https://docs.vmware.com/en/VMware-vSphere/6.0/com.vmware.vsphere.install.doc/GUID-F9056360-544A-4452-8C76-B29018235CEB.html (visited on 10/14/2021) (cit. on p. 19).

[63] Apache Foundation. *Configuring your CloudStack Installation*. URL: http://docs.cloudstack.apache.org/en/latest/installguide/configuration.html (visited on 10/14/2021) (cit. on p. 19).

[64] Google LLC. *Anthos clusters on bare metal quickstart*. URL: https://cloud.google.com/anthos/clusters/docs/bare-metal/1.6/quickstart (visited on 10/14/2021) (cit. on p. 19).

[65] ServiceNow Inc. *Day 1 setup guide for VMware on Cloud Provisioning and Governance*. URL: https://docs.servicenow.com/bundle/rome-it-operations-management/page/product/cloud-management-v2-setup/concept/cloud-mgt-vmware-setup-guide.html (visited on 10/14/2021) (cit. on p. 19).

[66] synetics GmbH. *VM Provisioning*. URL: https://kb.i-doit.com/display/en/VM+Provisioning (visited on 10/14/2021) (cit. on p. 19).

[67] Canonical Ltd. *How it works*. URL: https://maas.io/how-it-works (visited on 10/14/2021) (cit. on p. 19).

[68] Unkown. *What is Foreman?* URL: https://theforeman.org/introduction.html (visited on 10/14/2021) (cit. on p. 19).

[69] Unkown. *Introduction*. URL: https://wiki.fogproject.org/wiki/index.php?title=Introduction#What_is_FOG (visited on 10/14/2021) (cit. on p. 19).

[70] Thomas Lange. *FAI Guide (Fully Automatic Installation)*. URL: https://fai-project.org/fai-guide/#_a_id_work_a_how_does_fai_work (visited on 10/14/2021) (cit. on p. 19).

[71] Enno Gotthold. *Welcome to Cobbler's documentation!* URL: https://cobbler.readthedocs.io/en/latest/index.html (visited on 10/14/2021) (cit. on p. 19).

[72] Rackspace Technology Inc. *Welcome to Ironic's documentation!* URL: https://docs.openstack.org/ironic/latest/ (visited on 10/14/2021) (cit. on p. 19).

[73] Unkown. *What is Digital Rebar?* URL: https://rackn.com/rebar/ (visited on 10/14/2021) (cit. on p. 19).

[74] The Linux Foundation. *Architecture*. URL: https://docs.tinkerbell.org/architecture/ (visited on 10/14/2021) (cit. on p. 19).

[75] Microsoft Corporation. *Provision a Hyper-V host or cluster from bare metal computers*. URL: https://docs.microsoft.com/en-us/system-center/vmm/hyper-v-bare-metal (visited on 10/14/2021) (cit. on p. 19).

[76] Canonical. *Power management reference (snap/3.0/UI)*. URL: https://maas.io/docs/snap/3.0/ui/power-management (visited on 10/14/2021) (cit. on p. 19).

[77] ISPsystem LTD. *Server auto-add module*. URL: https://docs.ispsystem.com/dcimanager-admin/modules/server-auto-add-module (visited on 10/14/2021) (cit. on p. 19).

[78] Rackspace Technology Inc. *Welcome to VirtualBMC's documentation!* URL: https://docs.openstack.org/virtualbmc/latest/ (visited on 10/14/2021) (cit. on p. 20).

[79] netbiosX. *IPMI Default Password List*. URL: https://github.com/netbiosX/Default-Credentials/blob/master/IPMI-Default-Password-List.mdown (visited on 10/14/2021) (cit. on p. 20).

[80] Unkown. *Scripting*. URL: https://ipxe.org/scripting (visited on 10/14/2021) (cit. on p. 20).

[81] Unkown. *UEFI HTTP chainloading*. URL: https://ipxe.org/appnote/uefihttp (visited on 10/14/2021) (cit. on p. 20).

[82] 2pintsoftware. *Why is iPXE better that good old Plain Vanilla PXE?* URL: https://kb.2pintsoftware.com/help/why-is-ipxe-better-that-good-old-plain-vanilla-pxe (visited on 10/14/2021) (cit. on p. 20).

[83] Jean-Philippe Lang. *Using iPXE in Foreman*. URL: https://projects.theforeman.org/projects/foreman/wiki/Fetch_boot_files_via_http_instead_of_TFTP (visited on 10/14/2021) (cit. on p. 20).

[84] Jan-Piet Mens. *Network-booting machines over HTTP*. URL: https://jpmens.net/2011/07/18/network-booting-machines-over-http/ (visited on 10/14/2021) (cit. on p. 20).

[85] Tomaz Kosar, Nuno Oliveira, Marjan Mernik, Maria Pereira, Mateij Crepinsek, Daniela Cruz, and Pedro Henriques. *Comparing General-Purpose and Domain-Specific Languages: An Empirical Study*. Researchgate, 2010 (cit. on pp. 20, 27).

[86] Amazon Web Services Inc. *AWS CloudFormation concepts*. URL: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-whatis-concepts.html (visited on 10/14/2021) (cit. on p. 21).

[87] Amazon Web Services Inc. *What is the CloudFormation Command Line Interface (CLI)?* URL: https://docs.aws.amazon.com/cloudformation-cli/latest/userguide/what-is-cloudformation-cli.html (visited on 10/14/2021) (cit. on p. 21).

[88] Felix Richter. *Amazon Leads $150-Billion Cloud Market*. URL: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/ (visited on 10/14/2021) (cit. on p. 21).

[89] StackShare Inc. *AWS CloudFormation*. URL: https://stackshare.io/aws-cloudformation (visited on 10/14/2021) (cit. on p. 21).

[90] Amazon Web Services Inc. *AWS CloudFormation Linter*. URL: https://github.com/aws-cloudformation/cfn-lint (visited on 10/14/2021) (cit. on p. 21).

[91] Amazon Web Services Inc. *No title*. URL: https://console.aws.amazon.com/cloudformation/designer (visited on 10/14/2021) (cit. on p. 21).

[92] Rackspace Technology Inc. *Template Guide*. URL: https://docs.openstack.org/heat/rocky/template_guide/index.html (visited on 10/14/2021) (cit. on p. 22).

[93] Rackspace Technology Inc. *Heat architecture*. URL: https://docs.openstack.org/heat/latest/developing_guides/architecture.html (visited on 10/14/2021) (cit. on p. 22).

[94] Rackspace Technology Inc. *Orchestration service command-line client*. URL: https://docs.openstack.org/mitaka/cli-reference/heat.html (visited on 10/14/2021) (cit. on p. 22).

[95] Rackspace Technology Inc. *Python bindings to the OpenStack Heat API*. URL: https://docs.openstack.org/python-heatclient/latest/ (visited on 10/14/2021) (cit. on p. 22).

[96] Rackspace Technology Inc. *Understanding Bare Metal service*. URL: https://docs.openstack.org/ironic/latest/user/architecture.html (visited on 10/14/2021) (cit. on p. 22).

[97] Red Hat Inc. *About the Bare Metal Service*. URL: https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/15/html/bare_metal_provisioning/sect-introduction (visited on 10/14/2021) (cit. on p. 22).

[98] HashiCorp. *Providers*. URL: https://registry.terraform.io/browse/providers (visited on 10/14/2021) (cit. on p. 22).

[99] HashiCorp. *Providers*. URL: https://registry.terraform.io/browse/providers?category=infrastructure (visited on 10/14/2021) (cit. on p. 22).

[100] HashiCorp. *Write, Plan, Apply*. URL: https://www.terraform.io/ (visited on 10/14/2021) (cit. on p. 22).

[101] HashiCorp. *Plugin Development*. URL: https://www.terraform.io/docs/extend/index.html (visited on 10/14/2021) (cit. on p. 22).

[102] HashiCorp. *Configuration Syntax*. URL: https://www.terraform.io/docs/language/syntax/configuration.html (visited on 10/14/2021) (cit. on p. 22).

[103] HashiCorp. *Recovering from State Disasters*. URL: https://www.terraform.io/docs/cli/state/recover.html (visited on 10/14/2021) (cit. on p. 23).

[104] Pulumi. *Releases*. URL: https://github.com/pulumi/pulumi/tree/v0.3 (visited on 10/14/2021) (cit. on p. 23).

[105] Pulumi. *Pulumi*. URL: https://github.com/pulumi/pulumi (visited on 10/14/2021) (cit. on p. 23).

[106] Pulumi. *Cloud Providers*. URL: https://www.pulumi.com/docs/intro/cloud-providers/ (visited on 10/14/2021) (cit. on p. 23).

[107] Open Grid Forum. *Member Organisations*. URL: https://www.ogf.org/ogf/doku.php/members/organizational_members (visited on 10/14/2021) (cit. on p. 23).

[108] Open Grid Forum. *About*. URL: https://occi-wg.org/about/index.html (visited on 10/14/2021) (cit. on p. 23).

[109] Derrick Harris. *What's Next for OGF?* URL: https://www.hpcwire.com/2007/09/17/whats_next_for_ogf/ (visited on 10/14/2021) (cit. on p. 23).

[110] Deni Connor. *Consortium formed to promote enterprise grids*. URL: https://www.networkworld.com/article/2332351/consortium-formed-to-promote-enterprise-grids.html (visited on 10/14/2021) (cit. on p. 23).

[111] Open Grid Forum. *Implementations*. URL: https://occi-wg.org/community/implementations/index.html (visited on 10/14/2021) (cit. on p. 23).

[112] Open Grid Forum. *Home*. URL: https://occi-wg.org/index.html (visited on 10/14/2021) (cit. on p. 23).

[113] OASIS. *Standards*. URL: https://www.oasis-open.org/standards/ (visited on 10/14/2021) (cit. on p. 24).

[114] OASIS. *OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) Technical Committee*. URL: https://www.oasis-open.org/committees/membership.php?wg_abbrev=tosca (visited on 10/14/2021) (cit. on p. 24).

[115] OASIS. *OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC*. URL: https://www.oasis-open.org/committees/tosca/obligation.php (visited on 10/14/2021) (cit. on p. 24).

[116] OASIS. *Members*. URL: https://www.oasis-open.org/member-roster/ (visited on 10/14/2021) (cit. on p. 24).

[117] OASIS TOSCA Technical Committee. *TOSCA Version 2.0*. URL: http://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html (visited on 10/14/2021) (cit. on pp. 24, 31–33).

[118] OASIS. *OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC Public Documents*. URL: https://www.oasis-open.org/committees/documents.php?wg_abbrev=tosca (visited on 10/14/2021) (cit. on p. 24).

[119] Holger Reibold. *(German)*. URL: https://www.admin-magazin.de/Das-Heft/2018/02/Apache-ARIA-TOSCA (visited on 10/14/2021) (cit. on p. 24).

[120] VMware Inc. *TOSCA Components*. URL: https://docs.vmware.com/en/VMware-Telco-Cloud-Automation/1.9/com.vmware.tca.userguide/GUID-43644485-9AAE-410E-89D2-3C4A56228794.html (visited on 10/14/2021) (cit. on p. 24).

[121] OASIS TOSCA Technical Committee. *TOSCA Simple Profile in YAML Version 1.3*. URL: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html (visited on 10/14/2021) (cit. on p. 24).

[122] Cloudify Platform Ltd. *Official Plugins*. URL: https://docs.cloudify.co/latest/working_with/official_plugins/ (visited on 10/14/2021) (cit. on p. 25).

[123] Cloudify Platform Ltd. *Integrating with LDA*. URL: https://docs.cloudify.co/latest/working_with/manager/ldap-integration/ (visited on 10/14/2021) (cit. on p. 25).

[124] Dag Wieers. *Boot system using specific media through HP iLO interface*. URL: https://docs.ansible.com/ansible/latest/collections/community/general/hpilo_boot_module.html (visited on 10/14/2021) (cit. on p. 25).

[125] Felix Stephen and Anooja Vardhineni. *Configures the iDRAC network attributes*. URL: https://docs.ansible.com/ansible/latest/collections/dellemc/openmanage/idrac_network_module.html (visited on 10/14/2021) (cit. on p. 25).

[126] Dag Wieers. *Manage system objects in Cobbler*. URL: https://docs.ansible.com/ansible/latest/collections/community/general/cobbler_system_module.html#stq=netboot&stp=1 (visited on 10/14/2021) (cit. on p. 25).

[127] John Klein and Doug Reynolds. *Infrastructure as Code: Final Report*. Carnegie Mellon University, 2018 (cit. on p. 25).

[128] HashiCorp. *Terraform vs. CloudFormation, Heat, etc.* URL: https://www.terraform.io/intro/vs/cloudformation.html (visited on 10/14/2021) (cit. on p. 25).

[129] HashiCorp. *Terraform vs. Chef, Puppet, etc.* URL: https://www.terraform.io/intro/vs/chef-puppet.html (visited on 10/14/2021) (cit. on p. 25).

[130] Ian Buchanan. *How Infrastructure as Code (IaC) manages complex infrastructures*. URL: https://www.atlassian.com/continuous-delivery/principles/infrastructure-as-code (visited on 10/14/2021) (cit. on p. 25).

[131] Open Grid Forum. *OCCI in OpenStack*. URL: https://occi-wg.org/2012/07/18/occi-in-openstack/index.html (visited on 10/14/2021) (cit. on p. 26).

[132] Open Grid Forum. *OCCI in OpenNebula*. URL: https://occi-wg.org/2012/07/18/occi-in-opennebula/index.html (visited on 10/14/2021) (cit. on p. 26).

[133] Thijs Metsch. *OCCI for OpenStack*. URL: https://github.com/tmetsch/occi-os (visited on 10/14/2021) (cit. on p. 26).

[134] stackforge. *stackforge/occi-os*. URL: https://github.com/stackforge/occi-os (visited on 10/14/2021) (cit. on p. 26).

[135] Thijs Metsch, Andy Edmonds, and Marcin Spoczynski. *Occi*. URL: https://wiki.openstack.org/wiki/Occi (visited on 10/14/2021) (cit. on p. 26).

[136] Rackspace Technology Inc. *Welcome to the Heat documentation!* URL: https://docs.openstack.org/heat/latest/ (visited on 10/14/2021) (cit. on p. 26).

[137] Naveneetha Vasudevan and Laurence Tratt. *Comparative Study of DSL Tools*. Elsevier, 2011 (cit. on p. 27).

[138] Herbert Stachowiak. *Allgemeine Modeltheorie (German)*. Springer, 1974 (cit. on p. 27).

[139] HashiCorp. *The Core Terraform Workflow*. URL: https://www.terraform.io/guides/core-workflow.html (visited on 10/17/2021) (cit. on p. 30).

[140] HashiCorp. *State*. URL: https://www.terraform.io/docs/language/state/index.html (visited on 10/17/2021) (cit. on p. 30).

[141] HashiCorp. *Purpose of Terraform State*. URL: https://www.terraform.io/docs/language/state/purpose.html (visited on 10/17/2021) (cit. on p. 30).

[142] HashiCorp. *Remote State*. URL: https://www.terraform.io/docs/language/state/remote.html (visited on 10/17/2021) (cit. on p. 30).

[143] mirceal and jugg1es. *Hacker News*. URL: https://news.ycombinator.com/item?id=19472485 (visited on 10/17/2021) (cit. on p. 30).

[144] JetBrains. *DevOps*. URL: https://www.jetbrains.com/lp/devecosystem-2019/devops/ (visited on 10/17/2021) (cit. on p. 34).

[145] owulveryck. *owulveryck/toscalib*. URL: https://github.com/owulveryck/toscalib/network/members (visited on 10/17/2021) (cit. on p. 39).

[146] European Computer Manufacturers Association. *ISO 9660 Information processing - Volume and file structure of CD-ROM for infomration interchange*. 1988 (cit. on p. 41).

[147] John Davidson. *Minimal Linux Live*. URL: https://minimal.idzona.com/#home (visited on 10/20/2021) (cit. on p. 41).

[148] Alpine Linux Development Team. *How to make a custom ISO image with mkimage*. URL: https://wiki.alpinelinux.org/wiki/How_to_make_a_custom_ISO_image_with_mkimage (visited on 10/20/2021) (cit. on p. 41).

Eigenständigkeitserklärung

"Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.
Alle sinngemäß und wörtlich übernommenen Textstellen aus der Literatur bzw. dem Internet wurden unter Angabe der Quelle kenntlich gemacht."

Ort, Datum

Unterschrift