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 Matriculation number	Till Hoffmann 3135572
for the Supervisor	Daimler TSS GmbH Benjamin Gotzes
First advisor Second advisor	Prof. Dr. rer.nat. Stefan Traub Prof. Dr.-Ing. Philipp Graf

Submitted on 2021-10-28

Abstract

Infrastructure as Code (IaC) is a trend that applies software development techniques to infrastructure. Most currently available tools around it require a (cloud) backend that is available at all times and listens on API-calls like “I want X servers to be provisioned as Y”. But these backends have to be provisioned somehow as well. And at some level, hardware needs to be provisioned. A simple “Just give me a server X with Y” does not work here - or does it? This thesis tries to apply such IaC principles to hardware.

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

SAN Storage Area Network

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 do not scale in the range of tens or hundreds of nodes but in tens of thousands [1], [2] [3]. 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 (for example “add a node when the average CPU-load of all nodes exceeds 80 percent”) but the frequency for triggers is high. To simplify the management of nodes for both the cluster software and administrations, each of the nodes should 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 completely new field in computer science [4]. To be able to increase the number of components that 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. Abstraction layers like virtualization provide a partial solution for such cases.
But virtualization only provides an API on a single host; To be scalable and to be able to distribute new workloads efficiently (for example putting a new Virtual Machine (VM) on the hypervisor with the lowest load), an orchestrating software is needed. Examples of such software are VMware’s vSphere, OpenStack but also (formerly Google’s) Kubernetes.
These tools are capable of automatic live-migrations of workloads like VMs to distribute the load more equally and provide APIs for their features.
Another category for such orchestrators is public cloud providers such as Amazon Web Services (AWS), Microsoft Azure, and Google 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 the responsibility of the developer team of the latter to manage the underlying hardware (like 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 and therefore incompatible. 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 or more

specifically 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 introduces terms required to understand the topic. Then it describes the current state of the art of IaC and provisioning to identify issues and where compatibility makes the most sense. Afterward, different languages to describe IaC will be compared and the best fitting one selected. Before the architecture of a reference implementation of such a tool can be discussed, the final constraints and goals for it will be determined. Finally, the results are analyzed, so the questions from above can be answered.

2 Background

Searching online for IaC quickly leads to 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, must never go down (even for maintenance), are “hand fed” (changes are applied manually), and it is not feasible to redeploy them [6]. The term “cattle” is used when referring to machines that are never directly interacted with. Instead, all administrative interactions with them are automated. The approach of treating machines as cattle aims to standardize and therefore reduce the overall administrative effort for large amounts of servers. Standardization is also a requirement for automation. When operating on such a larger scale, 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 replaceable by design, which is not the case for pet machines. But even before those terms were introduced, some data centers were already too large to maintain each server manually. This chapter will guide through a part of the 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 data centers, 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 a foothold. KVM had one set of IO devices like monitor and keyboard attached on one side and several servers on the other side. Pressing the 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 corresponds 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 a 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 for 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 simultaneously. For data centers with more machines, this type of management does not scale very well (even financially, since

those 64-port switches tend to cost as much as a new car [7]).
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 image is then just copied over to new machines. The other is closely related but has a different name for each operating system. Examples are "preseed" for Debian, "setupconfig" for Windows, and "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 the setup with the default installation medium, the answers are predefined in a special file. This file can then be baked into the installation medium or separately (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 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" containing its MAC address 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 the 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 - all while 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 that made it unresponsible for remote access (like SSH) but did not power down the machine either, again an administrator was required to physically attend the server and manually resolve the issue.
The next generation of servers (since 1998) had such a remote control integrated into their mainboard, also rendering KVM obsolete because this new method scales vertically: 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 similar to 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 machines completely remotely via network, making physical attendance only required for changing physical parts of the infrastructure. The aforementioned embedded chips are called Baseboard Management Controller (BMC) and the surrounding technology is often 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 different names for their specific toolset. Examples are (not needed further in this thesis) DRAC from DELL EMC, ILO from HPE, and IMM from IBM. Not only the product names are different: Many features have different names but are also 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. Since IPMI adds an attack vector to the device, having such a chip in consumer devices decreases its security.
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 (or even in space [12]) by giving them remote input/output capabilities as well as power cycle control and the ability to get hardware sensor information. Since every vendor has a slightly different approach to their LOM interface, it is almost impossible to automate it universally. The network-boot technologies around PXE on the other side are capable of automating a high number of servers in parallel but do not provide necessary Input/Output for debugging issues.
Yet, 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 the state-of-the-art tools for remote-server-administration for several years now, 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 separates different workloads on different machines. This has the advantage that for example powering down a web server does not impact a database server. At the same time, it has the downside that servers are not used efficiently: When the database has almost no load, but the web-server is close to its limit, the load cannot be distributed properly 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 [13], 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 [14]. Starting 2005, Intel and AMD added hardware virtualization support 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 VMware's ESX hypervisor even dates back to 2001 but evolved to its successor ESXi in 2004. The first version of the Linux kernel which contained 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, the virtualization of machines enables migrations of them to another host without downtime ("live-migration"), finally allowing them to evacuate hosts before maintenance work. The same feature also drastically improves disaster recovery capabilities [15].
But the use of hypervisors and clustering them for live-migration and other cross-node functionalities has downsides 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), license/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 [16] [17] [18]. And VMware claims, that 80 percent of all virtualized workloads run on VMware technology [17], whereas Statista estimates their share to only 20 percent [19].

2.3 Cloud

The term cloud describes a group of servers that are accessed over the internet and the services (for example databases) that run on those servers [20]. These servers are located in one or (most often) multiple data centers. 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 consists of public clouds, which refers to publicly available services (i.e. shared with other organizations) [21]. 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, and [22]].
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 [23]. The current capacity exceeds even the market capitalization of Norway [24]. Considering the amount of revenue generated (at least $40 billion [24], it is obvious why the likes of Microsoft (in 2010) and Google (in 2013) followed Amazon into the cloud market [25].

Cloud computing can generate these high rates of revenue because they take advantage of economy of scale, very efficient sharing of resources, as well as a focus of a huge amount of developer effort into a relatively low amount of features (in contrast to every organization implementing the same feature set over and over for themselves) [26].
Apart from financial and developer efficiency, clouds have a long list of advantages and disadvantages [27], but these are out of scope here.
The high degree of automation and possibilities for scaling within a cloud environment made it possible to automate the process of this scaling. The time required to provision (and remove) new nodes plays an important role during autoscaling. This is where containers come in.

2.4 Containers

While the idea of containers exists for quite some time already (2006 as so-called cgroups, 2007 with LXC, [28] [29], it only reached mainstream popularity with the release of docker in 2013 [30]. The main difference between a VM and a container is the kernel: The former has a dedicated kernel, which runs in parallel with the hypervisor 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 [31].
Containers also (almost completely) solve the "works-on-my-machine" syndrome, where the developer machine is different from (for example) the production system to the extent 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 [32] [33] [34] [35] (or maybe the future of container computing looks different then previously thought [36] [37]).
Docker Inc. also introduced a cross-machine management tool called Swarm, which allows users to describe the desired state, which the engine tries to turn into reality (consistently). 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 services 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) [38]. 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 [34].
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: The birth of IaC.

2.5 Infrastructure-as-Code

IaC is the result of multiple factors:

- Software development encompasses more than running it, for example, a build pipeline, testing, and compliance. All of this has to be documented.
- Documentation is hard to hold up to date [39] [40]. This is not special to orchestrators or cloud providers but is true for all software.
- The only source of information that cannot lie (as in being out of date) is the source code.
- Scaling (infrastructure) leads to standardized objects.
- To have multiple instances of the same type of nodes, they have to be provisioned the same.
- The only (reliable) way to do 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 [41]. 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 [42] [43], the advantages gained by using IaC grow steadily. As soon as a certain 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 (for example commanding a hypervisor to create a VM). The other is a pull-based mechanism, where those subsystems (like a hypervisor) periodically ask the orchestrator whether tasks have to be completed [44].
These mechanisms not only apply to the interaction between the orchestrator and the subsystems but between the source code and the orchestrator as well.
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 of this is the hypervisor that acts as an intermediate layer between hard- and software-defined machines. The deployment (and configuration) of that middleware 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 [45]. Robots could 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 [45]. These range from being able to install the OS in an unattended way, over scriptable settings (or better: Non-scriptable ones - looking at you Windows) to compatibility with widespread instance initialization methods like cloud-init [46].
Another major difference in bare metal are the firmwares. Since they dictate the available features and how the interface of the hardware looks like API, it is important to have them in the correct version [45].

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 [47]. Even among those the domains they can (and want to) describe vary a lot. Additionally, 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 [48].
In contrast to a General-Purpose Language (GPL) (not to be confused with the license), its domain-specific counterpart promises higher success rates even with less experience and significantly higher closeness of mapping [49]. 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 [41].
Another differentiating characteristic 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, for example, "I want X instances of Y". When using the latter, it is the job of the orchestrator to compare the current state against the described desired state and conclude the required actions themselves [27]. Because IaC always aims at describing the whole state, declarative languages are better fitted for this task [41]. They also have the property of being idempotent: If applied multiple

times, the result does not change [41]. In order to describe the (whole) state of infrastructure, the declarative way is also more intuitive. It is the same way humans would describe a state (for example "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 like XML, JavaScript Object Notation (JSON), or Yet Another Markup Language (YAML) [41]. This includes both sub- and supersets of them. Libraries are internal DSLs as well [50]. "External" DSLs on the other hand are not directly related to other languages [50]. An example is the HashiCorp Configuration Language (HCL) used by Terraform [41] [50].
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 their 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 GPL, a DSL allows better separation of infrastructure code from other code [48]. Additionally, they are more context-driven, which makes them easier to work with for domain experts and users [50]. Their syntax is smaller and well-defined too, which makes them less complex as well [50].
By using a modeling language, much of the complexity behind the DSL is abstracted away. It is instead shifted to the technology experts that create those languages and the surrounding tools [50].
In an ideal world, a DSL for IaC is not a limiting factor; For example, it is not limited to full usage of only 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 license and owner of the language; It should not be left in the hands of a single organization, but a group (of several independent organizations/individuals). While a single owning organization tends to reflect itself in the software [51], 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.

3.1 State-of-the-art automated hardware provisioning

The interest in IaC has been increasing on a steady level over the last years [52]. It is estimated that ninety percent of global enterprises will rely on hybrid cloud by 2022 [53]. Surely boosted by the COVID-19 pandemic, 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 [54].
Instead of updating deployed instances, recreating them ensures all of them are equal [45]. 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 firmware or other "under-the-hand" changes [45].
There are differnt opinions on whether a local version of the state should be cached. Some state that it is a necessary means to improve performance [55], while others state that it is better not to assume certain states but instead check them [45]. In case of the latter, whenever a detected state is unexpected, the automation can exclude this certain node and tell the responsible humans to check what's wrong. The most reliable source of truth for the current state is the current state itself - not some kind of cached or partial version of it [45].
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 hardware with the correct drivers and OS preinstalled [56]. 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 [56]. The preinstalled OS makes the vendor (in this case Microsoft) responsible for provisioning. So it doesn't solve the problem but shifts it somewhere 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 (and provides no public documentation around it). Instead Amazon dispatch their own service personnel for every necessary manual task [57] [58].
Google doesn't have a product to bring its whole cloud on-premise or to the edge

yet, but only dedicated feature sets like the Google Kubernetes Engine. For it, the company relies on an underlying VMware vSphere environment and therefore outsources hardware management [59].

So how does the deployment of VMware's vSphere clusters work? The most important thing with the management software called vSphere Server 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 [60]), the hypervisor OS developed by VMware [61] [62]. In other words, vSphere requires (at least) one manually installed ESXi server, which can then host the vSphere Server software, which then has a feature called Auto Deploy [63]. This feature creates a PXE boot infrastructure that requires an external DHCP server [64] [65]. The latter has to be configured to distribute network boot details that point to the preexisting vSphere Server [64]. To reduce deployment time, Auto Deploy does not install the ESXi OS on machines but loads the boot image directly into its memory [66]. This implies that server restarts are handled equally as redeployments.
Considering VMware develops most of their software themselves (or owns a company that does it for them), the reliance on a third-party product in this field is surprising. 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 - its documentation starts after the hypervisor is installed [67].
One of the most recently published cluster software for bare metal is Google's Anthos. The software and its documentation completely omit the provisioning part up to the point where nodes can only be added when they are already accessible via SSH [68].
Common asset management tools like servicenow or i-doit use providers like vSphere or public clouds for instantiation [69] or even do not support hardware provisioning [70] at all.
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 [71] [72] [73] [74] [75] [76] [77] [78] [79]. 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 reading sensor data), RackNs Digital Rebar and ispsystems DCImanager [80] [76] [77] [81]. The main issues with using IPMI for provisioning are due to is its vendor-specific implementations. Not only is it not available for all hardware, but different vendors support different features (and versions) 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 [82].

Another reason for not using IPMI is its historically low security. Although most vendors each had their own credentials for accessing the management interface, they used the same combination of user and password for all of their devices [83]. Only with senate bill 327 chapter 886 (1798.91.04) taking effect in January 2020, the vendors must now use a unique random password for each machine. Most hardware manufacturers make a huge part of their revenue in the United States or even have their headquarter there. They have since deployed this feature as default for orders worldwide.
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 has its 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. As described in earlier chapters, iPXE is a very advanced and customizable network boot loader. For one, it is scriptable [84] - the script can even reside on a network location. Therefore it is very flexible in its configuration 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 significantly 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 [85] [86] [87] [88].
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 few properties change, it is feasible to use command-line arguments to describe the desired state for IaC tools [41]. But with a growing amount of properties the state-space increases, requiring a better way to describe it: Configuration files. The languages used within those files are mostly DSLs.

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 to gather the most prominent options. Because there are many DSLs, languages need to be ruled out based on their limitations afterward. As an intermediate result, two or three languages should remain. These can then be compared on a deeper level, for example, whether their internal design allows it 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 [89]. 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` [89]. Instead of requiring a command-line tool (there exists one [90] though), CloudFormation is designed to work by just uploading the file containing the definition. Possible sources are s3-buckets, git-repositories or manual uploads. Redeploys after changes have to be triggered manually. It is implied 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 [91] and was the first public cloud provider. CloudFormation is, therefore, one of the earliest DSLs for describing infrastructure. It is widely used [92], and the language itself as well as the tools around it can be assumed to be very mature. There are plugins for most IDEs [93]. 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 [93]. Under the hood, the linter uses schema validation. Assuming the schema is as mature as the language, it can be reasoned that this guarantees the 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 [94]. 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 definitely makes 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 [95]. When the API for CloudFormation is used, the Heat component translates the AWS-specific types to OpenStack compatible ones [96].
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 [96]. The contextual jargon (f.e. "stack") is also inspired by CloudFormation.
Another similarity is 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 [97] and an official library in/for python [98].
OpenStack supports bare metal via a component called Ironic [99]. It is the closest implementation of what this thesis desires to accomplish [100]. 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 [52]. 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 [101]. Of those providers "only" 160 are aimed at IaC [102]. Terraform uses HCL as DSL and is highly plugin-based [103] [104]. In addition to the custom language, JSON is supported as well [105].
Working with Terraform happens with a command-line executable. The binary loads plugins as needed, communicates with the APIs of the necessary providers, and gives feedback to the user [106]. To be as efficient as possible, Terraform maintains a local state file [55]. 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 [107]. Afterward, 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 [108]. 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 [109]. Another reason is the difficulty of recovering from such state disasters [109].

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" [110] [111]. Instead of using 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 [112]. This renders it basically useless for the scope of this thesis.

3.2.5 Open Cloud Computing Interface

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) in 2011 [113]. Their organizational member list mirrors their mainly academic purpose [114]. 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 [115]. 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) [115]. The OGF is backed by companies such as Dell EMC, NetApp and Oracle [116] [117]. Since the launch of the standard, several open-source cloud providers have started to support it, including OpenStack, OpenNebula, and CloudStack [118]. The last update of the specification was in 2016 [119]. The IT worlds changes fast, so five years since the last update is 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 organisation is well-known for widespread standards like Advanced Message Queuing Protocol (AMQP), Message Queuing Telemetry Transport (MQTT), OpenDocument (used as file format by OpenOffice and LibreOffice), PKCS#11 (a public-key cryptography standard), Security Assertion Markup Language (SAML) (an XML-based Single Sign-On standard and alternative to OpenID Connect), SARIF (standard format for static code analysis results) and VirtIO (the main I/O platform of KVM) [120]. 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 [121]

[122] [123]. The latest contribution was only one week before the time of writing, so its actively pursued and developed [124] [125].
TOSCA has been used in some proof-of-concept projects [27] in 2019, but their results were disappointing: The interfaces between the core standard and the supported providers are described as always out of date, which made 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 [126]. Still, TOSCA has a lot of plugins for platforms like OpenStack, VMWare, AWS, Google Compute Platform (GCP) and Microsoft Azure (Azure) (obviously strongly related to the earlier mentioned member organizations), configuration management tools like ansible, chef, puppet and saltstack or container orchestrators like docker swarm and kubernetes [127] [126], [128]. All those projects conclude that the standard is extremely promising, but the current state makes it impossible to use properly [126].
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 specification of a file archive format called Cloud Service ARchive (CSAR). These archives contain five major parts [129]:

- 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

The original TOSCA specification has a strong focus on how things should work together but does neither give complete examples nor describe how applications should be designed. Therefore, in addition to the TOSCA standard itself, OASIS also published an extending standard called "TOSCA Simple Profile" [130]. While large parts of both specifications are redundant, the Simple Profile provides the types and ecosystem needed for real-world applications of TOSCA. Examples are types for Compute, Storage, and Credentials. 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 [129].

3.2.7 OASIS TOSCA with Cloudify

Cloudify is another implementation of an TOSCA orchestrator, but instead of supporting the Simple Profile, it uses another extension - also named Cloudify. This means that the Cloudify extension provides twin types to the likes of the aforementioned Compute, Storage, and Credentials. Both TOSCA implementations / extensions of the TOSCA standard are closely related. even their approach is similar: Both have their orchestrator implemented as the backend of a web application.

Both allow visualization and (partial) graphical editing of the CSAR definitions. Both provide 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 [131]. It is very well documented and has many useful integrations like LDAP for authentication and authorization [132].

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 [133] [134]. But for "general" bare-metal provisioning it mostly relies on external systems like cobbler [135].

3.2.9 Others

There are a lot of DSLs around IaC. Many of them were not introduced here. The main reason is that their purpose is different, their approach doesn't fit the use case, or they are too specialized. 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 [136] [41] [137] [138] [139].
These two categories are named differently in different sources, for example using "Infrastructure Definition" as a synonym for provisioning and "Configuration Registry" for configuration management. Some software like Ansible can also fill both roles [41].
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.

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 [140] [141] [142] [143]. 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 [27], OCCI will not be looked at in too much detail as well. For example, the documentation for the OpenStack implementation [140] recommends to visit the corresponding wiki-site, which is even older [144].
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 [145]. Additionally, there are few examples, and to run it even in a proof-of-concept style requires installing and running many components of OpenStack. These requirements make it not only unfeasible 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 a 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 out 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 supporte
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 [146] [49] [27] [147]. They compare based on attributes like (no specific order):

- Primary approach [146]
- Guarantees provided in case of well-formedness [146]
- Reusability of components [146]
- Error proneness and reporting like line number and column offset [146] [49]
- Efficiency: Amount of code for a given case study [146]
- Aspects to learn for a given case study or how hard the mental operations are [146]
- Viscosity: How hard it is to make changes/updates [49]
- Hidden dependencies like requiring agents, a dedicated server or a third-party software [49]
- Visibility: How easy is it to find the responsible snippet in the codebase [49]
- 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 take 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 reasonable amount of time).
It is important to note that it is out of this thesis’ scope to compare the languages on a deeper level, 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 can display the same amount of information more densely. 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 a language on its own, 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 Terraform executable detects necessary plugins out of the provided code and downloads them automatically from a repository like the Terraform registry.

Still, the plugin system is a curse 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 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 every infrastructure integration can be described. But the quality of the ecosystem depends on the quality of the plugins in most cases. Since Terraform's 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. For example, some providers only have to provide VMs, while others provide managed clusters of specific software (databases, Kubernetes, ...).
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 [148]. The authoring part is self-explaining. The planning is done automatically by the Terraform executable [148]. It compares the state described in the code with the current state and creates 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) it is used to significantly improve the performance of the planning phase [55]; The local mirroring of the actual state enables Terraform to work with the state as fast as a local file-access instead of requiring several API-requests. This is especially convenient in large environments, where the provisioned state consists of hundreds or thousands of instances [107]. 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 [107] in order to prevent race conditions when deploying. If Terraform is integrated into 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 using a so-called remote state backend that provides state locking as the main feature [149]. Supported backends range from a shared folder over Terraform Cloud to S3-compatible object storage.
It is strongly advised against manipulating the state manually [55]. 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 state file) can result in almost unrecoverable crashes of Terraform [150].
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.

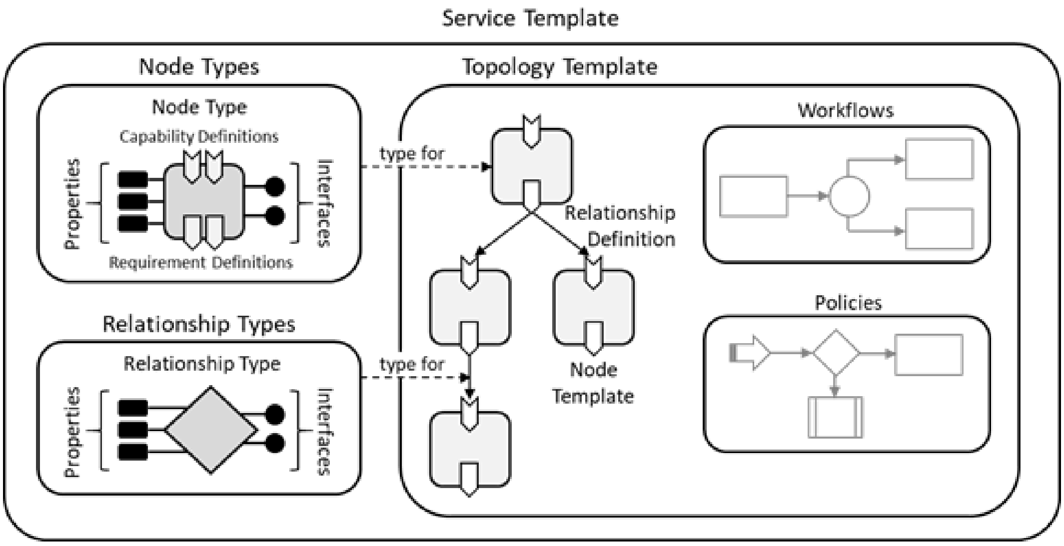

Figure 3.1: *Architecture of TOSCA and the components of CSAR files [124]*

Because the parts (for example a web application on a web server 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 described 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 web applications 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. To reflect where relationships are allowed (for example a web application 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 3.2 displays how types and templates correspond and how requirements and capabilities are mapped.

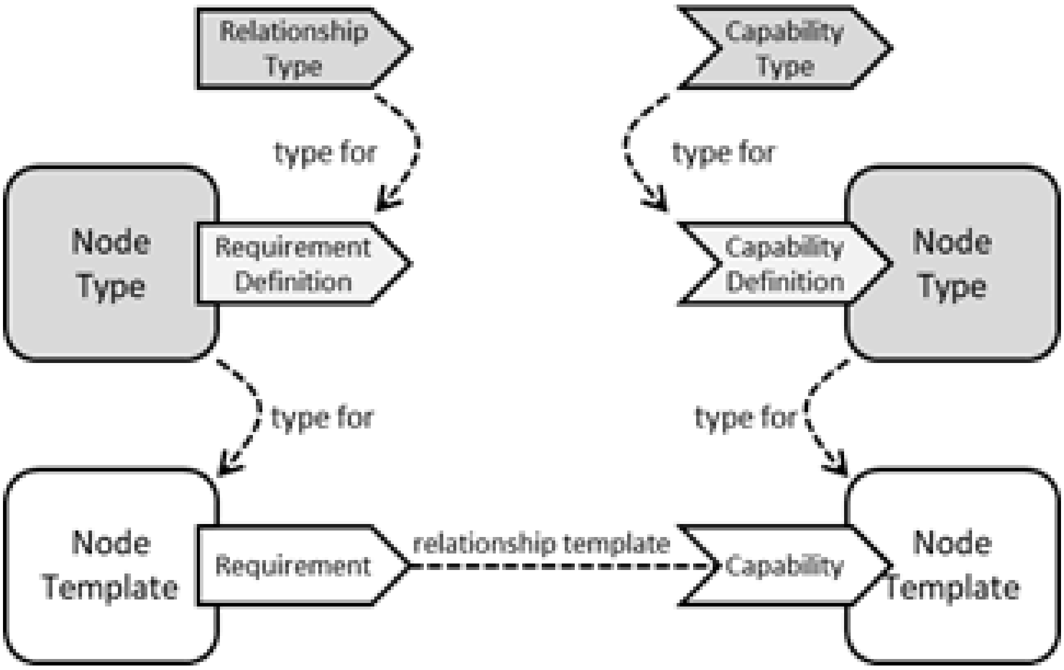

Figure 3.2: *Derivation and relationships between TOSCA elements [124]*

The last part contained in CSAR files are 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 [124]. 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 the 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 3.3 visualizes this substitution approach.

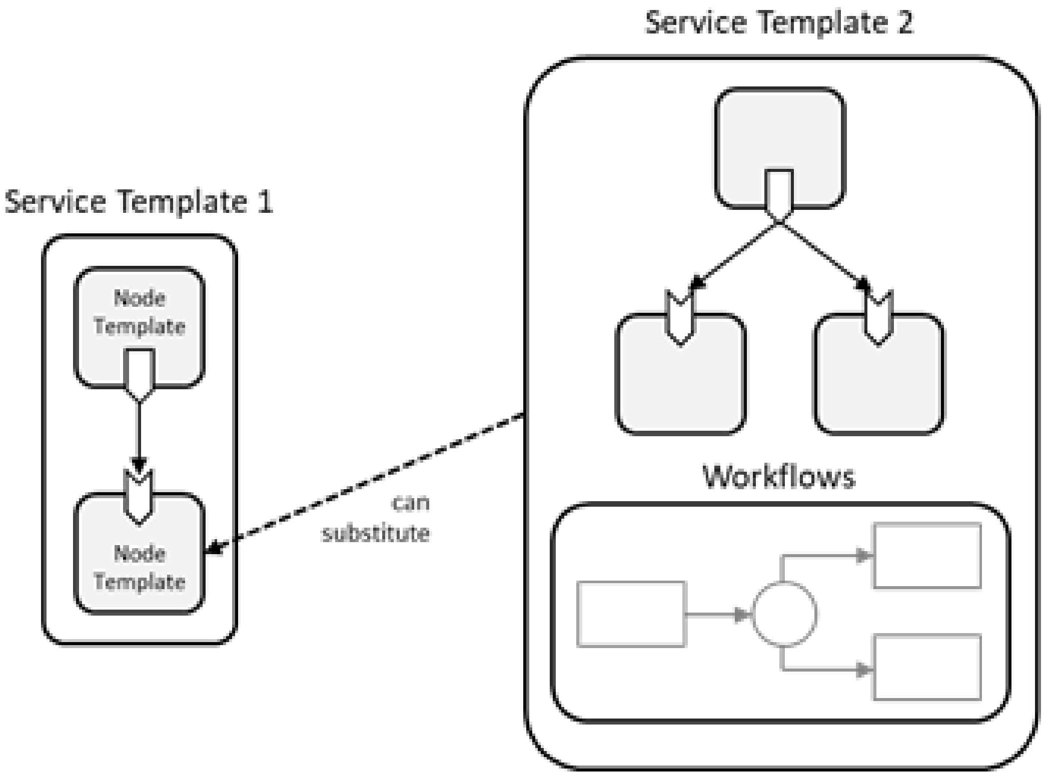

Figure 3.3: *Substitution of node templates with external topologies [124]*

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 command-line 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 number of initial candidates and the length of the selection process, deciding 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 number 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 feasible 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 developer base, so all development on it distributes across many organizations.
As of 2021, Terraform is the state of the art: It supports more providers than any organization should want to use simultaneously, has the largest market share, and is considered by most providers as a key player [151]. 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

As can be seen by the earlier description of provisioning tools, most of them offload the lowest level of it - the hardware provisioning - to other tools. This chapter tries to determine up to which level the provisioning is needed in order to be integratable with such tools.

3.6.1 Necessarity of VMs

Since the rise of containers, they are often compared to VMs. Most comparisons can be summarized as follows: "Containers are super fast (to start and stop), have less overhead, and are better for scaling. Running them on a bare-metal system is optimal (speedwise at least). Yet, VMs provide a better isolation level of their workloads. So it depends on your use case."
Some of those articles have titles like "Are VMs dead?" [152] [153]. Apart fromt the speed, overhead and isolation level differences, there are two additional characteristics that (still) justify the existance of VMs. One is the already mentioned live-migration capability. The ephemeral nature of containers requires applications to

be designed in a way that forgives container restarts during migrations. In case the application has a hard requirement where that this is not possible, a VM still is the best option. The other reason relates to the number of clusters and their size. The trend for containerized applications comes with a trend that many developers want/have/need multiple clusters for themselves in order to test the application in an as close as real environment as possible. Since the purpose of these clusters is container orchestration, they cannot themselves run in containers. The other two options are VMs and bare metal machines. While running the orchestration software on bare metal is perfectly feasable and allows to omit the virtualization layer, it removes the capability to chunk the machines into smaller nodes to use in multiple clusters. In the days of multi-CPU servers, it can not be efficient to give each developer its own dedicated cluster on bare-metal machines. And with bare-metal clusters, the minimum machine size defines the minimum cluster size. The cluster will not be used most of the day, the developer needs access to it, the hardware has to be managed somehow and the effort required for redeployment is unimaginable. As described earlier, all of this is the perfect use case for VMs. More than that, virtualization allows to distribute (cluster-)machines over different (fire) zones, datacenters, countries and even continents. Responsible for such capabilities is the orchestration software that integrates with all hypervisors and combines them into a cluster.

3.6.2 Deployment of applications

Deployment of containerized applications can hardly be compared to common applications. But since the containers run on applications like a container orchestration software or a container runtime as well, those can be seen as the common application equivalent.
Applications have many ways to be installed. Downloading an executable, installers in fileformats like *.deb, *.apk, *.snap, and *.msi, direct compilation from source code or with package managers like apt, zypper, and yum are just a few to have named some. These are definitely too many to be covered by this thesis, and most probably too many to be issued by a hardware provisioning extension. The least common denominator of all the ways to install applications is the console access - since that how they are installed under normal circumstances. The most common way to provide remote console access is SSH. So in order to deploy applications, the lowest level that is required by all ways to install applications is SSH.

3.6.3 Deploment of SSH

While SSH could be treated just like any application that needs to be installed/configured via console access, it is preinstalled in most (all?) OS distributions and the configuration could be completed before the installation of the OS. Embedding a fixed password in such an image would be really bad for security. But there is an alternative way, which is way more secure: Embedding only a public key into the image. More on this in a later chapter.

3.6.4 Deployment of the operating system

Since the OS provides the abstraction level and standardized interfaces applications want/need, it is required to provision a hardware machine. In a world without them, applications would contain lots of redundant code for device I/O and drivers would be even more of a mess than they already are.
The combination of SSH for rmeote console access with an OS that has SSH active and configured even before it is installed provides flexible installation methods and a standardized approach for hardware provisioning.

-

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 sub-tasks, 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. 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:
In 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 the network. Therefore, the orchestrator has to manipulate an external or internal (as in "integrated into the orchestration software") DHCP server. 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 it is transferred via the 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 like installing a package on or copying files to the machine. Optimally, the user should be informed of what is currently going on during the whole process.
Figure 4.1 shows the whole workflow in an interaction overview diagram.

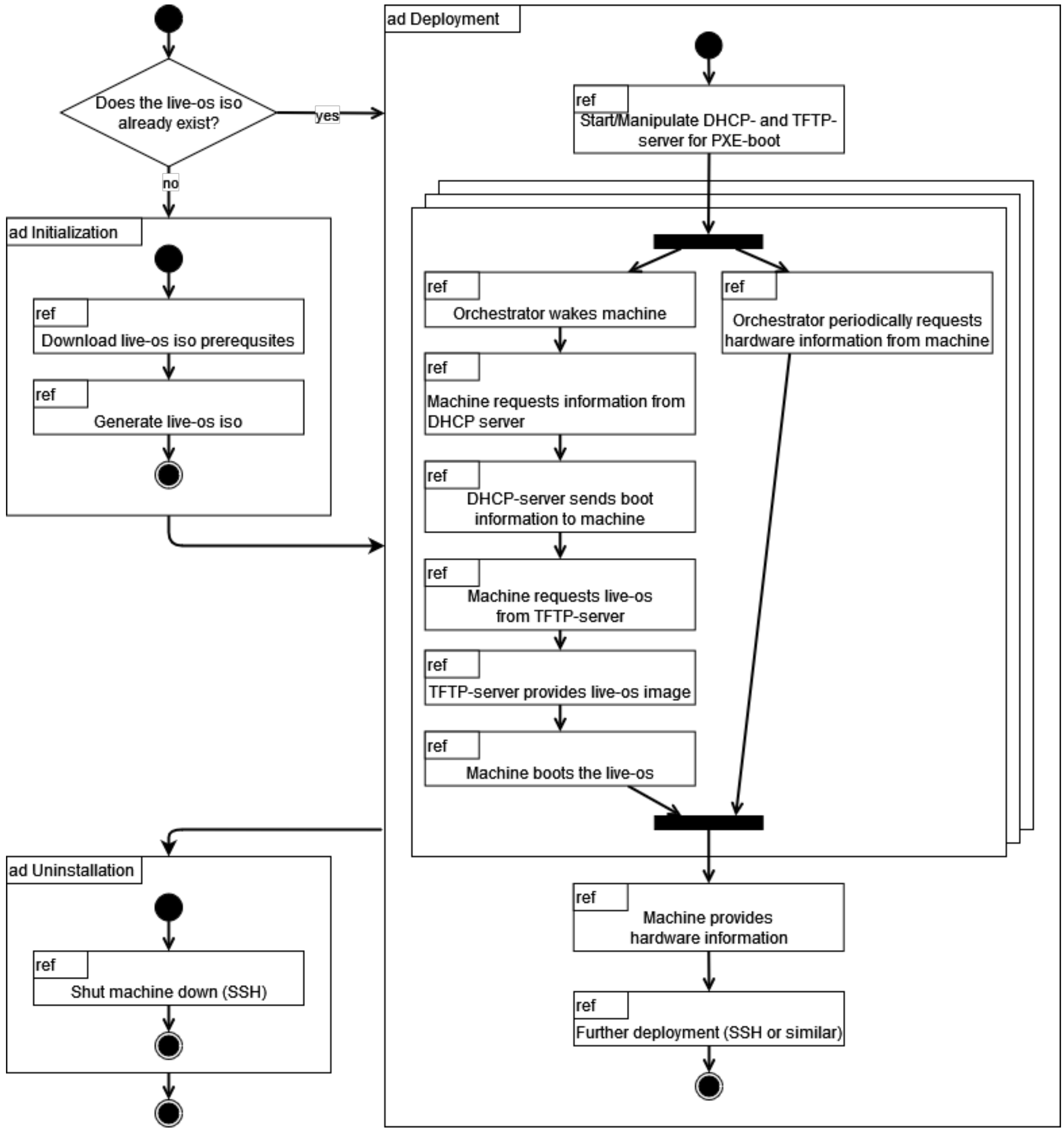

Figure 4.1: *UML interaction overview diagram describing the application workflow*

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 command-line-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 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.

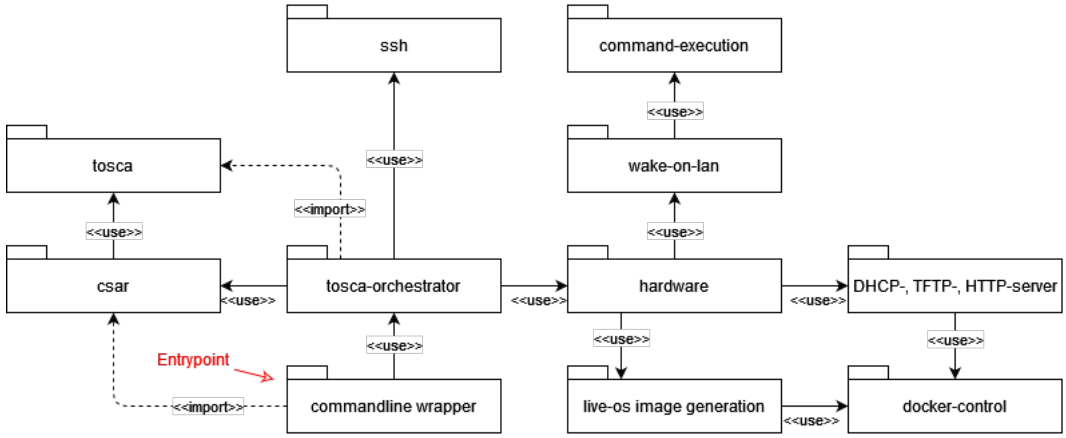

Figure 4.2: *UML package diagram describing how the packages are related to each other*

As can be seen in the diagram, the command-line 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 hardware package directly. The former two 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 Commandline-wrapper

Due to its nature as a wrapper, the sole purpose of this package is the translation of command-line arguments and flags into the correct call to methods of the orchestrator package. The package is also responsible for differentiating between subcommands and additional arguments. During that process, it also validates the mix of arguments and flags.
This package is the entry point of the whole application, and it translates what the user wants into what the orchestrator understands. Apart from this, its implementation is relatively unspectacular and mostly self-explaining.

4.3.2 Orchestrator

The orchestrator is the center of the application. It is getting called once per execution by the command line wrapper. Afterward, the orchestrator controls what happens during the runtime. In case of a new deployment, the orchestrator will first call the CSAR package with the file path of the CSAR file. The result is a fully parsed TOSCA instance (more on this in the other chapters). Depending on the contents, the orchestrator will execute commands on the local machine, remotely via SSH and/or make calls to the hardware package. Therefore, it is also responsible for conforming to dependencies between the components defined in the CSAR file. This includes as much parallelization as possible regarding the steps necessary to reach the desired state.
The TOSCA standard defines several conformance targets. Most of them refer to the files and their structure, two of them are the TOSCA processor and the TOSCA orchestrator. This package conforms to the latter. This means it contains a processor (the TOSCA package described next), can process CSAR files (with the CSAR package also described later), all TOSCA-internal functions defined by the specification and generates meaningful errors when necessary.

4.3.3 TOSCA

To be able to implement a library for the TOSCA standard, it is necessary to work through both its actual 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, both sources are often required.
Depending on the chosen programming language, such a package might already exist. In the case of this thesis' reference implementation Golang is the language of choice. Sadly, most libraries are either incomplete or based on one another - the most complete of them is severely outdated [154].
To support the latest version of the TOSCA and Simple-Profile specifications, and to be able to easily extend it if necessary, a complete reimplementation of the TOSCA

library was created. It is strongly influenced by the previously mentioned outdated one but is more complete (and up-to-date). Only with the already described line-by-line read-through of both specifications, it is possible to gather enough information about the standard, how it works, and to differentiate what is contained in the standard and what is just an example.

With the new library, it is possible to parse any (valid) TOSCA 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`, it validates that all desired features are implemented. While the specifications were unclear or simply incomplete on several occasions (more on this in the outlook, where possible improvements are discussed), further looks at the reference implementation of OpenTOSCA or the resulting reasonable assumptions helped in creating the library. Some incomplete passages were simply not required for the proof-of-concept this thesis tries to achieve and could therefore be left out during the implementation. The other cases, such as inconsistencies or required assumptions will be covered in the chapter “Outlook”.

Because the basic TOSCA specification describes how to work with extensions such as the Simple Profile, it is possible to implement the necessary import-, reference-, and dependency-system 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.

So, this package does not only parse YAML to TOSCA structures in native Golang but also resolves type derivations, imports, namespacing, and validates them all.

As described before, this package fulfills the requirements for a TOSCA processor. Per the definition, this means it parses and recognizes TOSCA elements, generates useful errors when necessary, validates requirements as stated in each type and template definition, resolves imports (while respecting the namespaces), and resolves the defined substitutions.

4.3.4 CSAR

After the TOSCA package can parse the contents of files, the file-I/O is implemented as a separate package. The last chapter of the specification (actually in the last chapters of both specifications - the actual standard and the Simple-Profile extension) contains information about how to pack multiple artifacts like OS-images, definition-, or other required files together into one CSAR file. The file is 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 separation is 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 existence, 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.

4.3.5 Command-execution

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 with the TOSCA specification.
The package provides 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.

4.3.6 Docker control

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 the 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 case, in particular, 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 anymore. And a second time, when the application terminates due to an error somewhere else and the default termination is not reached. Even “deferring” the container termination does not work. Only when the SIGTERM interrupt of the wrapper application is “manually” listened for and a function removes the container in such a case, the removal is successful in all cases.
Another obstacle is the retrieval of live logs during the container LiveCycle and embedding the retrieved output in the logs of the wrapping application. This can

be solved by creating a buffered stream reader, 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.

4.3.7 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. 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 a network, a virtual version of such a system is required. One such file format for disk images is defined by the ISO 9660 standard [155]. 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.
To be able to create an ISO file that provides information about the underlying hardware from scratch, only one requirement should exist: 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 [156]. A first test with the original downloaded iso file seemed promising: The file size is less than 50 MB, the OS boots in under 30 seconds and 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 web server is started when the system boots. But with or without this extension, the generated image is not bootable on the said hypervisor. Tested were both the BIOS and UEFI firmwares - to no avail.
A second promising OS is Alpine Linux. It is a full-blown operating system with a focus on a 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 [157]. Since there are additional applications necessary to be installed to generate the image, and these applications 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 does not have as many features (preinstalled). Therefore its initial size is significantly lower and its non-tweaked 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, enabling 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 completely 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.

4.3.8 DHCP-, TFTP-, HTTP-server

The DHCP-server does not only provide devices with (temporary) IP addresses but additional information about the network as well. Common examples are the subnet mask and the network gateway for routing beyond the current network. But the DHCP-server can advertise options for network-boot as well. This includes the URL of a boot firmware on a TFTP-server. Since BIOS and UEFI require different

files there, a condition is required to run on the DHCP-server. As described earlier, TFTP is not very efficient, and to support HTTP, a boot firmware named iPXE is distributed and loaded before the actual operating system. The loaded iPXE will then send another DHCP request and will as a result begin to load iPXE anew. To escape this loop, the DHCP-server can detect whether the client is iPXE or not. In case it is not, the iPXE firmware for this specific client is provided (as described above). In case it is, a script location is passed. This can be a simple HTTP URL. The HTTP-server can then dynamically provide the client with special iPXE-instructions and the actual live-OS image.
The software dnsmasq combines DHCP and TFTP server (DNS as well, but that is not needed here). To start and stop the servers on-demand and in one go, dnsmasq and a webserver of choice are installed in a container image. When running the container, the host port 67 (UDP) is required for DHCP-, host port 69 (UDP) for TFTP-, and host port 80 (TCP) for the HTTP-server. The previously generated live-OS image is provided as a volume mount.

4.3.9 Wake-on-lan

To wake a machine, a so-called magic packet is broadcasted. This packet has a very special format. The only "real" payload is the MAC-address of the NIC that can power the attached machine when the packet is received.
This package contains methods for the creation and broadcasting of the magic packet for a MAC address. Since WOL is incompatible with most hypervisors, an additional mechanism has to check whether a hypervisor hosts a VM with the specified MAC-address. Should this be the case, it determines its identifier and issues the hypervisor to power the VM on.

4.3.10 Hardware

The hardware package manages the bare metal provisioning process. It controls the container that is responsible for the DHCP-, TFTP- and HTTP-servers. It also initiates the live-OS image generation. Additionally, it issues the calls to the Wake-on-LAN package. Because loading and booting the live-OS takes several seconds, a simple HTTP request would time out and the provisioning would fail. Therefore, this package also manages the timeouts and retries until the hardware information is available. Because of the (relatively high - minutes or hours instead of seconds) lease time of issued IP addresses, the DHCP-server assigns different IP addresses to a machine that wants to boot via network, and when the live-OS comes up. The hardware package therefore also checks the lease-list of the DHCP-server for the latest IP address for each MAC address.
Upon having gathered all desired information about the hardware, the package returns the information to the orchestrator package.

The orchestrator somehow needs to understand when (and how) to use the hardware package. This is accomplished with an additional TOSCA extension similar to the Simple-Profile one. The lowest level node of the standard is the `tosca.nodes.Compute` node. It has several attributes like IP addresses and attached networks. Additionally, it provides a capability called `tosca.capabilities.Compute`. This capability is required by all `tosca.nodes.SoftwareComponent` nodes, which basically means every application that could potentially run on a machine. The `Compute` node only requires some kind of block storage. (So, to be accurate, block storage is the lowest level. But in the world of bare-metal, storage requires computing first.)
In order to be compatible with the original standard as much as possible, an alternative for the `Compute` node is defined. This `baremetal.nodes.Compute` node provides the exact same capabilities as its TOSCA counterpart. Since capabilities are matched via name only, this works seamlessly. In addition, the attributes `tosca.nodes.Compute` has, must be available, too. Additional attributes are possible as well, but they are not used by standard TOSCA applications. With them, information like desired firmware versions, hardware requirements (like at least two power-supply-units) and more could be defined in the CSAR file.
In an optimal case, there is a default OS. Then, in order to use bare-metal machines instead of virtual ones, it is sufficient to change the type of the underlying node from `tosca.nodes.Compute` to `baremetal.nodes.Compute`.
Currently, the TOSCA specification and the Simple-Profile extension do not describe many nodes that could be "implemented" in hardware. But some examples are `tosca.nodes.LoadBalancer` and the aforementioned `tosca.nodes.BlockStorage`. Many other devices could and should be supported by this hardware extension, but computing serves well as a proof-of-concept. Routers, switches, or even a Storage Area Network (SAN) are potential candidates for such further "devices". Hardware parts are also an option: That way, graphic cards, multiple processors, and multiple NICs could become node selectors.

4.3.11 SSH

To provide the most possible security for deployed live-OS images, SSH password authentication is disabled. To run remote commands, an SSH public key is included in the image generation workflow. This package manages the generation of a public/private key pair, as well as invoking commands on the remote machines. In the case of the latter, it returns the output to the orchestrator for further processing (like logging).

5 Analysis

The currently described software architecture does not aim at a complete installation of an OS on a machine but instead uses the live-OS deployed during the information-gathering phase. Additional work is necessary to support not only the unattended installation of an OS, or even several different ones.

Another bottleneck of the current design is that all machines have to be booted to gather information on the underlying hardware. All machines are started simultaneously, producing one huge network traffic and power spike. In the test runs for the proof-of-concept implementations, this was not an issue, because the machines were only a few. In larger environments, including larger data centers, this approach is not an option. Therefore, the machines should be started in batches, where the batch size might scale with the current load of the issuing host (as the capacity hugely depends on the network bandwidth). Alternatively, a viable solution for bare-metal systems with no other orchestrator might be to provision machines dedicated for further the handling of DHCP, TFTP and HTTP. The number of these can then be scaled separately. While this could partially solve the network bottleneck, the power scaling problem persists nevertheless.

The aforementioned dedicated services needed for provisioning are not necessarily fully managed and integrated into the described orchestrator. Interfaces for external systems like networking hardware should be implemented and integrated.

Even apart from these services, a plugin system similar to the one provided by Terraform would bring huge benefits. While TOSCA has a similar feature with its imports, namespaces, and substitutions, the orchestrator itself needs additional capabilities as well. This includes the hardware extension described in this thesis, but also applies to the original TOSCA standard as well: Currently limited to Bash and Python, plugins for additional implementation artifact languages are thinkable.

The currently described provisioning workflow requires an initial list of MAC addresses. Further research should investigate whether retrieval of those addresses could be automated. A first idea would be a network broadcast, which leads to the addresses being communicated via the ARP protocol.

Apart from the implementation details on the tool itself, the TOSCA standard or at least its specification has improvement potential as well. On several occasions, the YAML examples provided along with the introduction of a new structure are invalid. The issues range from YAML-structure errors, inconsistencies or simply missing information. Table 5.1 lists those in the order they occur in the specification. The Simple-Profile extension has similar issues, those are listed in table 5.2.

The specification would also profit greatly from a detailed description of a reference orchestrator like the one by OpenTOSCA (which already is the semi-official reference implementation). That way, the tasks of the orchestrator would be clearer,

Table 5.1: *List of issues with the TOSCA specification*

Chapter	Issue description
4.2.6.2.7.2	Indentation error
4.3.5.6.3.3	Indentation error
4.4.2	Paths used in `get_property` should adhere to the format of `[<entity_name>, <optional_req_or_cap_name>, <property_name>, <nested_property_name_or_index>*]`.
4.4.7.2	It is unclear whether `external_schema` is a string or complete schema definition.
4.5.5.2	Unclear whether the properties and attributes are a list or a map. (Here they are lists, in all other occurences they are maps.)
5.2.1	Indentation error
5.3.1.3	Missing output name makes the examples grammar invalid. It is not possible to efficiently detect whether `optional_req_or_cap_name` or another `nested_property_name_or_index` is set.

some hidden design decisions in the standard would be at least described somewhere close and the development of such orchestrators would be significantly easier.

Assuming a scenario where a new datacenter should be provisioned from scratch (no OS deployed), the tool described in this thesis can automate the initial setup. Because TOSCA is the language of choice, and a compatible orchestrator is embedded in the application, users can make use of all capabilities the DSL provides. Since the original language can already be used for further provisioning and configuration management, the resulting application is a mixture of hardware provisioning, IaC provider, and -client.
The resulting application is not only another IaC tool with TOSCA as its chosen language. It is also the first tool that can provision bare-metal machines on demand. And together with the hardware extension, it is one of the view tools, where description and deployment of virtual and physical infrastructure as code with the same language are possible.
In a scenario where infrastructure is already deployed, and with a DSL different than TOSCA, it is required to translate/migrate the codebase at least partially. But since TOSCA allows integrations of other DSLs and tools, this can happen step by step. The same applies to existing infrastructure without any IaC codebase.
The included TOSCA orchestrator fulfills all requirements described in the specifications of the standard and its Simple-Profile extension. Because the hardware extension requires low-level access to some host capabilities (like listening on host port 67 and 68), the application needs to run with extended privileges.

Table 5.2: *List of issues with the Simple-Profile specification*

Chapter	Issue description
2.1	Inconsistent value for `tosca_definitions_version`.
3	Redundant with TOSCA specification.
unkown	The Simple-Profile definition of "Operation Implementation" has an attribute `operation_host`, while in the TOSCA specification it does not.
unkown	"operation implementation" and "notification implementation" of the Simple-Profile specification are completely distinct, while in the TOSCA specification they are treated as if they are the same.
unkown	The "interface definition" keynames do not contain the field "type", but it is listed in grammar notations. The TOSCA specification complies the former.
5.3.4.2.2	Inconsistent indentation (3 spaces)
5.3	Inconsistent datatype naming; Whilst case sensitive, some names start with a lowercase and others with an uppercase letter ("Root", "json", "xml", "Credential").
5.3.6.5.1	Indentation error
5.3.6.5.4	Followed by 5.3.6.5.6, so 5.3.6.5.5 is missing.
5.3.8 - 5.3.10	Do not contain information on whether any field is required.
5.4	The description states there are three categories of artifacts. Listed are 4.
5.7.1.2	Contains indentation error and `derive_from` is not defined.
5.8.3	First chapter to contain a type definitions, earlier subchapters do not - this is inconsistent.
5.8.4.2 till 5.8.4.4	Incomplete chapters and the only provided diagram is incorrect. (State after deletion is "configured", which is wrong.)
5.8.5	Description text is copied from 5.8.4 and does not apply here. Also, missing descriptions on properties and attributes.
5.8.5.2	Missing illustration.
5.9.1.3	Indentation error
5.9.8	Databases can only have one username-password combination. Additionally, there is a dedicated Credential type, but it is not used here. (Also observed in other occurrences.)
5.9.9.2	Definition of capabilities is undefined.
5.9.10	Default constraints for storage instances are `greater_or_equal: 0.1 MB` (or `1 MB` or `1 GB`). Better: `greater_than: 0B`.

6 Conclusion

At the beginning of this thesis, it was unclear, whether bare-metal machines can be deployed on-demand and without an always-on operator. With the hardware extension for TOSCA and the introduced command-line tool, it is now clear that this is possible.

The extension also showed how hardware could be modeled in an IaC language and how it blends in with the existing definitions.

Additionally, the hardware detection step with the live-OS made it clear that requirements regarding the hardware (like the minimum amount of RAM required for a certain application) can be fulfilled at runtime, too.

The command-line tool approach that the thesis took was initially meant to leave things simple, but as it turned out, it has many benefits over the always-on approach other tools have. Not only can it just run on-demand, but it doesn't require any kind of (infrastructure-)bootstrapping before it can get to work.

On one side, its on-demand nature and the low initial requirements set this tool greatly apart from other hardware-provisioning tools. On the other side, making the ability to describe and provision hardware part of the orchestrator with an admittedly relatively common IaC capability set it apart from other IaC tools. The result is something that did not exist previously and has the potential to initiate a new breed of tools that broaden the infrastructure part of IaC.

The TOSCA standard was found to be extremely flexible and easy to integrate with other tools - which are not necessarily part of the orchestrator. The two Golang packages around CSAR and TOSCA are also mentionable successes along the way and might help the standard to increase its userbase.

During the DSL finding phase, additional dimensions were developed that are important when comparing such languages. While the listed dimensions help select a language, this thesis made it also clear that they are (still) not the only means required to be looked at for choosing the right language for specific use cases.

While working with the TOSCA specification, several smaller issues in the document itself were found. But the work also highlighted some larger issues about the design of the standard, that would drastically simplify it and decrease the initially required learning curve.

7 Outlook

Apart from the "obvious" improvements described in the Analysis chapter, other improvements would increase the usability of TOSCA: For one, the standard currently strongly differentiates between "attribute" fields and "property" fields. According to the specification, attributes reflect the actual state of the entity during its lifecycle once instantiated. Properties on the other hand are used to describe the desired state. In the real world, those terms are often used as synonyms for each other. Therefore the current approach is extremely confusing, and new users will most probably confuse them several times. Since TOSCA already has other means in place to distinguish the desired state from the current state (like types, templates, and instances), it should only be an implementation detail within the orchestrator how it separates them. The separation also introduces additional complexity during derivations and makes it hard to retrieve the values because of the choice between the functions `get_propery` or `get_attribute`.

By allowing external artifacts and only describing what should be done with those, TOSCA natively allows extending it with additional DSLs. While there currently does not exist such a case, it is for example possible to describe an API with the OpenAPI (formerly Swagger) DSL - designed with the single goal to describe APIs. As already stated elsewhere, there are many DSLs, each with its use case. TOSCA allows to integrate them as they are. This is a huge benefit, but such integrations are missing yet. Future work could look at such languages and develop integrations with TOSCA.

The long list of DSLs specifically for IaC makes clear that there is not a single one that fits them all. In most cases, this can be traced to subtle different features between the providers. Further comparing the different DSLs and searching for common ground could help in developing an industry-wide standard. In a perfect world, migrations between infrastructure providers are done by exchanging the provider interface and the credentials. As is done with other standards as well, organizations can research, try, and develop their own products of course. But as soon as there are several providers, and migrations are necessary, those transitions should be as easy as possible. Not only the users would benefit: The providers that collaborate on such a standard encourage third party software to adhere to it as well, making them compatible and therefore increasing the potential userbase. A common standard would also distribute necessary work on API design across all backing organizations. Another reason for a standard is its long-term lifecycle; Creating meta-models for the DSLs (models describing the language) become more feasible, which might then lead to more reusability, and finally to better tooling around development and migrations (from and to other languages).

In addition to the improvements on the language and standard side, interactions with bare-metal machines would profit from another iteration of interfaces. It is a bit strange that a protocol like TFTP is still the state of the art for transmitting boot images in network boot environments in 2021. Vendors should bake support for HTTP or HTTPS into their BIOSes and UEFI-systems. They could even use iPXE as their default network boot firmware (so they do not need to reinvent the wheel).

IPMI brings many convenient features like remote firmware upgrades, and API for sensor data, and the ability to change boot settings remotely. This includes information about the hardware like product ID, serial number, and firmware version. Yet, information about the hardware like the amount of RAM is often not accessible directly. Having such a feature would render the relatively complex hardware detection step with the live-OS described in this thesis obsolete - significantly decreasing the required time and resources during this step.

By leveraging IPMI, the hardware provisioning introduced in this thesis can be extended even more: Up- and downgrading firmware automatically, setting the boot-order and other BIOS and UEFI configurations, as well as controlling the power cycle in a more direct way (forced power-off for example) are just a few examples.

Another feature that is currently not available, is setting a network boot URL (like the HTTP-URL) to the live-OS image directly, without requiring additional settings on the DHCP level.

This goes hand in hand with another problem concerning BMCs and IPMI. Currently, the naming of features, including many available API calls regarding firmware differ greatly between all vendors. This makes it impossible to write such tools around those interfaces, that support all vendors equally at the same time. Instead, vendor support often has to be built one at a time - making it an quite unpleasant journey.

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. 7).

[2] Mark Baker. *Cluster Computing White Paper*. University of Portsmouth, 2001 (cit. on p. 7).

[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. 7).

[4] Till Hoffmann. *Schemavalidation of YAML-files in the Context of Kubernetes & Helm*. Technische Universität Ulm, 2018 (cit. on p. 7).

[5] Martin Fowler. *Martin Fowler: Snowflake Servers*. URL: `https://dzone.com/articles/martin-fowler-snowflake` (visited on 10/13/2021) (cit. on p. 9).

[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. 9).

[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 pp. 9, 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. 10).

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

[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. 10).

[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. 10).

[12] Matt Monson. *SpaceX AMA - We are the SpaceX software team, ask us anything!* URL: `https://old.reddit.com/r/spacex/comments/gxb7j1/we_are_the_spacex_software_team_ask_us_anything/ft6jcdf/` (visited on 10/26/2021) (cit. on p. 11).

[13] Shannon Meier. *IBM Systems Virtualization: Servers, Storage, and Software*. Redpaper, 2008 (cit. on p. 11).

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

[15] 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. 12).

[16] 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. 12).

[17] VMware Inc. *Accelerate IT. Innovate with Your Cloud.* VMware Inc, 2012 (cit. on p. 12).

[18] 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. 12).

[19] Thomas Alsop. *Share of the global server market in the first half of 2018 and 2019, by virtualization type*. URL: https://www.statista.com/statistics/915091/global-server-share-physical-virtual/ (visited on 10/26/2021) (cit. on p. 12).

[20] 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. 12).

[21] 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. 12).

[22] 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. 12).

[23] 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. 12).

[24] 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. 12).

[25] 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. 12).

[26] 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).

[27] Valeriya Shvetcova, Oleg Borisenko, and MAxim Polischuk. *Domain-specific language for infrastructure as code*. IEEE, 2019 (cit. on pp. 13, 15, 23, 25, 26).

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

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

[30] 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. 13).

[31] Camino Cielo Cloud Services. *VPS Provisioning Times*. URL: https://www.vpsbenchmarks.com/labs/provisioning_times (visited on 10/14/2021) (cit. on p. 13).

[32] 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. 13).

[33] Constance Drugeot. *Is serverless the future?* URL: https://www.devopsonline.co.uk/is-serverless-the-future/ (visited on 10/14/2021) (cit. on p. 13).

[34] 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. 13).

[35] 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. 13).

[36] 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. 13).

[37] 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. 13).

[38] 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. 13).

[39] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. *How software engineers use documentation: the state of the practice*. IEEE, 2003 (cit. on p. 14).

[40] C. J. Satish and M. Anand. *Software Documentation Management Issues and Practices: a Survey*. Redpaper, 2016 (cit. on p. 14).

[41] Kief Morris. *Infrastructure as Code, second Edition*. O'Reilly Media, Inc., 2008 (cit. on pp. 14–16, 19, 24).

[42] Ranjit Bawa and Rick Clark. *Software-defined everything - Breaking virtualisation's final frontier*. Deloitte University Press, 2008 (cit. on p. 14).

[43] M. N. O. Sadiku1, S. R. Nelatury, and S.M. Musa1. *Software Defined Everything*. Journal of Scientific and Engineering Research, 2017 (cit. on p. 14).

[44] 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. 14).

[45] Rob Hirschfeld. *Can Infrastructure as Code apply to Bare Metal?* RackN, 2021 (cit. on pp. 15, 17).

[46] Canonical Ltd. *cloud-init Documentation*. URL: `https://cloudinit.readthedocs.io/en/latest/` (visited on 10/14/2021) (cit. on p. 15).

[47] Juan Cadavid, David Lopez, Jesús Hincapié, and Juan Qintero. *A Domain Specific Language to Generate Web Applications*. Researchgate, 2009 (cit. on p. 15).

[48] Thorsten Berger. *Domain-Specific Languages (DSLs) motivation, concepts, examples*. University of Gothenburg, 2018 (cit. on pp. 15, 16).

[49] 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. 15, 26).

[50] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 2010 (cit. on p. 16).

[51] Melvin E. Conway. *Conway's Law*. URL: `http://www.melconway.com/Home/Conways_Law.html` (visited on 10/14/2021) (cit. on p. 16).

[52] 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. 17, 21).

[53] 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. 17).

[54] 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. 17).

[55] HashiCorp. *State*. URL: `https://www.terraform.io/docs/language/state/index.html` (visited on 10/17/2021) (cit. on pp. 17, 21, 29).

[56] 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. 17).

[57] Amazon Web Services Inc. *AWS Outposts*. URL: `https://aws.amazon.com/outposts/` (visited on 10/14/2021) (cit. on p. 17).

[58] Amazon Web Services Inc. *AWS Outposts FAQs*. URL: `https://aws.amazon.com/outposts/faqs/` (visited on 10/14/2021) (cit. on p. 17).

[59] 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. 18).

[60] 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. 18).

[61] 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. 18).

[62] 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. 18).

[63] 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. 18).

[64] 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. 18).

[65] 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. 18).

[66] 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. 18).

[67] 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. 18).

[68] 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. 18).

[69] 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. 18).

[70] synetics GmbH. *VM Provisioning*. URL: https://kb.i-doit.com/display/en/VM+Provisioning (visited on 10/14/2021) (cit. on p. 18).

[71] Canonical Ltd. *How it works*. URL: https://maas.io/how-it-works (visited on 10/14/2021) (cit. on p. 18).

[72] Unkown. *What is Foreman?* URL: https://theforeman.org/introduction.html (visited on 10/14/2021) (cit. on p. 18).

[73] Unkown. *Introduction*. URL: https://wiki.fogproject.org/wiki/index.php?title=Introduction#What_is_FOG (visited on 10/14/2021) (cit. on p. 18).

[74] 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. 18).

[75] Enno Gotthold. *Welcome to Cobbler's documentation!* URL: https://cobbler.readthedocs.io/en/latest/index.html (visited on 10/14/2021) (cit. on p. 18).

[76] Rackspace Technology Inc. *Welcome to Ironic's documentation!* URL: https://docs.openstack.org/ironic/latest/ (visited on 10/14/2021) (cit. on p. 18).

[77] Unkown. *What is Digital Rebar?* URL: https://rackn.com/rebar/ (visited on 10/14/2021) (cit. on p. 18).

[78] The Linux Foundation. *Architecture*. URL: https://docs.tinkerbell.org/architecture/ (visited on 10/14/2021) (cit. on p. 18).

[79] 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. 18).

[80] 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. 18).

[81] 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. 18).

[82] Rackspace Technology Inc. *Welcome to VirtualBMC's documentation!* URL: https://docs.openstack.org/virtualbmc/latest/ (visited on 10/14/2021) (cit. on p. 18).

[83] 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. 19).

[84] Unkown. *Scripting*. URL: https://ipxe.org/scripting (visited on 10/14/2021) (cit. on p. 19).

[85] Unkown. *UEFI HTTP chainloading*. URL: https://ipxe.org/appnote/uefihttp (visited on 10/14/2021) (cit. on p. 19).

[86] 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. 19).

[87] 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. 19).

[88] 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. 19).

[89] 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. 20).

[90] 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. 20).

[91] 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. 20).

[92] StackShare Inc. *AWS CloudFormation*. URL: https://stackshare.io/aws-cloudformation (visited on 10/14/2021) (cit. on p. 20).

[93] Amazon Web Services Inc. *AWS CloudFormation Linter*. URL: https://github.com/aws-cloudformation/cfn-lint (visited on 10/14/2021) (cit. on p. 20).

[94] Amazon Web Services Inc. *No title*. URL: https://console.aws.amazon.com/cloudformation/designer (visited on 10/14/2021) (cit. on p. 20).

[95] Rackspace Technology Inc. *Template Guide*. URL: https://docs.openstack.org/heat/rocky/template_guide/index.html (visited on 10/14/2021) (cit. on p. 21).

[96] Rackspace Technology Inc. *Heat architecture*. URL: https://docs.openstack.org/heat/latest/developing_guides/architecture.html (visited on 10/14/2021) (cit. on p. 21).

[97] 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. 21).

[98] 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. 21).

[99] 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. 21).

[100] 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. 21).

[101] HashiCorp. *Providers*. URL: https://registry.terraform.io/browse/providers (visited on 10/14/2021) (cit. on p. 21).

[102] HashiCorp. *Providers*. URL: https://registry.terraform.io/browse/providers?category=infrastructure (visited on 10/14/2021) (cit. on p. 21).

[103] HashiCorp. *Write, Plan, Apply*. URL: https://www.terraform.io/ (visited on 10/14/2021) (cit. on p. 21).

[104] HashiCorp. *Plugin Development*. URL: https://www.terraform.io/docs/extend/index.html (visited on 10/14/2021) (cit. on p. 21).

[105] HashiCorp. *Configuration Syntax*. URL: https://www.terraform.io/docs/language/syntax/configuration.html (visited on 10/14/2021) (cit. on p. 21).

[106] HashiCorp. *Plugin Development*. URL: https://www.terraform.io/docs/extend/index.html (visited on 10/26/2021) (cit. on p. 21).

[107] HashiCorp. *Purpose of Terraform State*. URL: https://www.terraform.io/docs/language/state/purpose.html (visited on 10/17/2021) (cit. on pp. 21, 29).

[108] HashiCorp. *How Terraform Works With Plugins*. URL: https://www.terraform.io/docs/extend/how-terraform-works.html (visited on 10/26/2021) (cit. on p. 21).

[109] HashiCorp. *Recovering from State Disasters*. URL: https://www.terraform.io/docs/cli/state/recover.html (visited on 10/14/2021) (cit. on p. 21).

[110] Pulumi. *Releases*. URL: https://github.com/pulumi/pulumi/tree/v0.3 (visited on 10/14/2021) (cit. on p. 22).

[111] Pulumi. *Pulumi*. URL: https://github.com/pulumi/pulumi (visited on 10/14/2021) (cit. on p. 22).

[112] Pulumi. *Cloud Providers*. URL: https://www.pulumi.com/docs/intro/cloud-providers/ (visited on 10/14/2021) (cit. on p. 22).

[113] Ralf Nyr en, Andy Edmonds, Alexander Papaspyrou, and Thijs Metsch. *Open Cloud Computing Interface - Core*. Open Grid Forum, 2011 (cit. on p. 22).

[114] Open Grid Forum. *Member Organisations*. URL: https://www.ogf.org/ogf/doku.php/members/organizational_members (visited on 10/14/2021) (cit. on p. 22).

[115] Open Grid Forum. *About*. URL: https://occi-wg.org/about/index.html (visited on 10/14/2021) (cit. on p. 22).

[116] 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. 22).

[117] 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. 22).

[118] Open Grid Forum. *Implementations*. URL: https://occi-wg.org/community/implementations/index.html (visited on 10/14/2021) (cit. on p. 22).

[119] Open Grid Forum. *Home*. URL: https://occi-wg.org/index.html (visited on 10/14/2021) (cit. on p. 22).

[120] OASIS. *Standards*. URL: https://www.oasis-open.org/standards/ (visited on 10/14/2021) (cit. on p. 22).

[121] 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. 22).

[122] 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. 23).

[123] OASIS. *Members*. URL: https://www.oasis-open.org/member-roster/ (visited on 10/14/2021) (cit. on p. 23).

[124] 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. 23, 30–32).

[125] 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. 23).

[126] Holger Reibold. *(German)*. URL: https://www.admin-magazin.de/Das-Heft/2018/02/Apache-ARIA-TOSCA (visited on 10/14/2021) (cit. on p. 23).

[127] Tobias Binz, Uwe Breitenbücher, Oliver Klopp, and Frank Leymann. *TOSCA Topology and Orchestration Specification for Cloud Applications*. University of Stuttgart, 2021 (cit. on p. 23).

[128] 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. 23).

[129] Tim Waizenegger. *How to Build CSARs for OpenTOSCA*. University of Stuttgart, 2021 (cit. on p. 23).

[130] 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. 23).

[131] Cloudify Platform Ltd. *Official Plugins*. URL: https://docs.cloudify.co/latest/working_with/official_plugins/ (visited on 10/14/2021) (cit. on p. 24).

[132] 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. 24).

[133] 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. 24).

[134] 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. 24).

[135] 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. 24).

[136] John Klein and Doug Reynolds. *Infrastructure as Code: Final Report*. Carnegie Mellon University, 2018 (cit. on p. 24).

[137] HashiCorp. *Terraform vs. CloudFormation, Heat, etc.* URL: https://www.terraform.io/intro/vs/cloudformation.html (visited on 10/14/2021) (cit. on p. 24).

[138] HashiCorp. *Terraform vs. Chef, Puppet, etc.* URL: https://www.terraform.io/intro/vs/chef-puppet.html (visited on 10/14/2021) (cit. on p. 24).

[139] 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. 24).

[140] 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. 25).

[141] 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. 25).

[142] Thijs Metsch. *OCCI for OpenStack*. URL: https://github.com/tmetsch/occi-os (visited on 10/14/2021) (cit. on p. 25).

[143] stackforge. *stackforge/occi-os*. URL: https://github.com/stackforge/occi-os (visited on 10/14/2021) (cit. on p. 25).

[144] Thijs Metsch, Andy Edmonds, and Marcin Spoczynski. *Occi*. URL: https://wiki.openstack.org/wiki/Occi (visited on 10/14/2021) (cit. on p. 25).

[145] Rackspace Technology Inc. *Welcome to the Heat documentation!* URL: https://docs.openstack.org/heat/latest/ (visited on 10/14/2021) (cit. on p. 25).

[146] Naveneetha Vasudevan and Laurence Tratt. *Comparative Study of DSL Tools*. Elsevier, 2011 (cit. on p. 26).

[147] Herbert Stachowiak. *Allgemeine Modeltheorie (German)*. Springer, 1974 (cit. on p. 26).

[148] HashiCorp. *The Core Terraform Workflow*. URL: https://www.terraform.io/guides/core-workflow.html (visited on 10/17/2021) (cit. on p. 29).

[149] HashiCorp. *Remote State*. URL: https://www.terraform.io/docs/language/state/remote.html (visited on 10/17/2021) (cit. on p. 29).

[150] mirceal and jugg1es. *Hacker News*. URL: https://news.ycombinator.com/item?id=19472485 (visited on 10/17/2021) (cit. on p. 29).

[151] JetBrains. *DevOps*. URL: https://www.jetbrains.com/lp/devecosystem-2019/devops/ (visited on 10/17/2021) (cit. on p. 33).

[152] Ian Hood. *Where are my Containers? Are VMs dead?* URL: https://www.redhat.com/en/blog/where-are-my-containers-are-vms-dead (visited on 10/26/2021) (cit. on p. 33).

[153] Dan Kusnetzky. *The Death of Virtual Machines? Not so Fast*. URL: https://virtualizationreview.com/articles/2014/11/05/death-of-virtual-machines.aspx (visited on 10/26/2021) (cit. on p. 33).

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

[155] European Computer Manufacturers Association. *ISO 9660 Information processing - Volume and file structure of CD-ROM for infomration interchange*. 1988 (cit. on p. 42).

[156] John Davidson. *Minimal Linux Live*. URL: https://minimal.idzona.com/#home (visited on 10/20/2021) (cit. on p. 42).

[157] 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. 42).

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