Master Thesis

to obtain the degree „Master of Science“

Extending Infrastructure-as-Code to bare-metal

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

Submitted on 2021-10-31

Abstract

Bla/laber/fasel

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

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

SaaS Software-as-a-Service

SAML Security Assertion Markup Language

SSH Secure Shell

TFTP Trivial File Transfer Protocol

TOSCA Topology and Orchestration Specification for Cloud Applications

UEFI Unified Extensible Firmware Interface

VM Virtual Machine

WOL Wake On LAN

YAML Yet Another Markup Language

Table of contents

1 Introduction

Todays distributed applications don't scale in the range of tens or hundreds of nodes but in tens of thousands [Distributed Systems - Concepts and Design George Coulouris, Cluster Computing White Paper Mark Baker, https://cloud.google.com/blog/products/containers-kubernetes/google-kubernetes-engine-clusters-can-have-up-to-15000-nodes].
In order to be as fast and efficient as possible, the number of nodes has to automatically scale up and down based on their usage. The conditions are simple (f.e. „add a node when all nodes have reached 80 percent cpu load") but the frequency for triggers is high. to simplify management of nodes for both the cluster software and administrations, each node should also be set up the same way. A perfect use-case for automation.
The process of software-defining infrastructure is called Infrastructure-as-Code (IaC). Using software development tools to manage infrastructure has many more advantages like version-control, collaboration, reviews, automated tests and continuous deployment. The accompanying combination of development and operations called DevOps opened a complete new field in computer science [BA, chapter 2]. To be able to increase the amount of components than can be software-defined, the underlying hardware needs to support it. As an example, processors have a fixed architecture, while with Field-Programmable Gate Array (FPGA) chips it is configurable.

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

different. Second, these orchestrators all do mostly the same thing, but with different efficiency and flexibility. Third, each one of them has one initial requirement: Someone has to do the initial bootstrapping, i.e. somehow set up the orchestrator. Again, this does not solve the hardware management problem, but shifts it to a different problem which (hopefully) requires less effort to solve.
This thesis aims at three fundamental questions: Can bare-metal machines be deployed on-demand like virtual workloads on providers. Is it possible to do so without the requirement of an always-on operator, thus removing the initial bootstrapping effort. And last but not least, how can hardware constraints be mirrored in IaC languages. TODO should these be with dots or question marks? Should this be a numbered/unnumbered list?
The paper at hand first explains how workload provisioning historically evolved and introduce terms required to understand the topic. Then it describes the current state of the art of IaC and provisioning in order to identify issues and where compatiblity makes the most sense. Afterwards different languages to describe IaC will be compared and the most fitting one selected. Before the architecture of an example tool can be discussed, the final constraints and goals for it will be determined. A final discussion analyses the results and answer the initial questions.

2 Background

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

2.1 Bare-metal

In the early times of datacenters, they required quite the administrative effort. Reinstalling an operating system on a server required one administrator to be physically located close to the server, some kind of installation media, a monitor and at least a keyboard. Since both monitor and keyboard were rarely used, Keyboard, Video, Mouse (KVM) (not to be confused with the linux kernel virtual machine with the same abbreviation) quickly gained foothold. KVM had one set of IO-devices like monitor and keyboard attached on one side and several servers on the other side. Pressing a corresponding button, the complete set of IO-devices would be „automatically“ detached from whatever server it was previously connected to and attached to the machine the button refers to.
Those devices still exist and evolved into network-attached versions, which means they don't require administrators to press buttons on the device and instead of dedicated set of IO-devices per handful of servers, they allow administrators to use the ones attached to their workstation. So these devices introduce some kind of remote control for servers, including visual feedback. Their main issue is not the dedicated cabling they require to each server, but the limited amount of servers they can be attached to. The largest KVM-Switches have 64 ports https://kvm-switch.de/en/category-335/from-16-Port-KVM-Switches/64-Port-KVM-Switches/, meaning they can be attached to 64 machines. For datacenters with more machines, this type of management doesn't scale very well (even financially, since those 64-port switches

tend to cost as much as a new car).
Instead of installing each operating system manually, two methods for unattended installations emerged: One is the creation of so-called „golden images“, where all needed software is preinstalled, settings are baked in, correct drivers are in place and so on https://opensource.com/article/19/7/what-golden-image. The other is closely related and has a different name for each operating system. Examples are „preseed“ for debian, „setupconfig“ for windows, „cloud-init“ for various operating systems including ubuntu (2008, https://github.com/canonical/cloud-init/releases?after=ubuntu-0.3.1). Under the hood they all work the same: Instead of asking the user each question during setup, the answers are predefined in a special file. This file can be baked in into the golden image or seperately (even on-demand via network).
With those methods, administrators only need to attach the installation medium, configure the machine to boot from it and power-on the machine. While this does save a large amount of time already, it still requires manual interactions with the machine.
To further automate machine installations, technologies like Trivial File Transfer Protocol (TFTP) (1981), Preboot eXecution Environment (PXE) (1984), Bootstrap Protocol (BOOTP) (1985) emerged and concluded in the development of Dynamic Host Configuration Protocol (DHCP) (1993). Only when Intel released the Wake On LAN (WOL) in 1997 and PXE 2.0 as part of its Wired-for-Management system in 1998 it was possible to fully network-boot a device.
PXE uses DHCP to assign an ip-address to a Network Interface Card (NIC). When the NIC receives a so-called „magic packet“ during the WOL process, it triggers the machine to power-on. Depending on the basic input/output system (BIOS) or Unified Extensible Firmware Interface (UEFI) (which is a newer implementation of the former) settings, the machine starts with its configured boot-order, for network-boot this means an embedded Network Bootstrap Program (NBP) (for example the widespread network boot loaders PXELINUX or iPXE), which are like a networking equivalent to what GNU GRand Unified Bootloader (GRUB) and the Windows Boot Loader are for local disks: It downloads a kernel from a (network) resource, loads it into memory and finally (chain-)boots the actual Operating System (OS) [https://www.networxsecurity.org/de/mitgliederbereich/glossary/n/network-bootstrap-program.html https://docs.openstack.org/ironic/latest/user/architecture.html].
The combination of all those technologies finally allows to remotely power-on a machine, boot a kernel via network instead of a local disk, and using the NIC as the interface for those abilities, outsourcing the bootstrapping and scaling to the network infrastructure.
But there are still some issues with those technologies:
When a machine had an error which made it unresponsible for remote access (like SSH), but didn’t power the machine down neither, again an administrator was required to phyiscally attend the server and manually resolve the issue.
The next generation of servers (since 1998) had such a remote control integrated into their mainboard, rendering KVM obsolete, because this new method scales vertical: Every new server has an embedded chip that acts as an integrated remote control. Unifying those efforts into a single standard for the whole industry, Intel published a specification called Intelligent Platform Management Interface (IPMI)

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

2.2 Virtualization

Even though IBM shipped its first production computer system capable of full virtualization in 1966 [https://en.wikipedia.org/wiki/Hypervisor], 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. [Loftus, Jack (December 19, 2005). "Xen virtualization quickly becoming open source 'killer app"'. TechTarget. Retrieved October 26, 2015. -> http://searchdatacenter.techtarget.com/news/1153127/Xen-virtualization-quickly-becoming-open-so
Starting 2005, Intel and AMD added hardware virtualization to their processors and the Xen hypervisor was published. Other hypervisors followed: Microsoft Hyper-V and Proxmox Virtual Environment were both published in 2008. The initial release of VMwares ESX hypervisor even dates back to 2001, but evolved to ESXi in 2004. The first version of the linux kernel containing the Kernel-based Virtual Machine (KVM) hypervisor (not to be mistaken with the equal abbreviation for keyboard, video, mouse described earlier - from this point onwards, KVM always refers to the hypervisor) was published in 2007.
Apart from the previously stated advantages, virtualization allowed for live-migrations of machines to another host without downtime, finally allowing to evacuate a machine prior to maintenance work. The same feature also drastically improves disaster recovery capabilities [https://searchservervirtualization.techtarget.com/definition/server-virtualization
But the use of hypervisors and clustering them for live-migration and other cross-node functionalities has a downside as well: Vendor lock-in, since the different VM formats are not compatible (there are some migration/translation tools, but best practices for production environments advise against them), licence / support fees in addition to the hardware support fees and requiring additional expertise for the management software.
Yet, 100 percent of the fortune 500 and 92 percent of all business used (server-)virtualization technologies in 2019 [https://www.statista.com/statistics/1139931/adoption-virtualization https://www.vmware.com/files/pdf/VMware-Corporate-Brochure-BR-EN.pdf, https://www.spiceworks.com/marketing/reports/state-of-virtualization/].

2.3 Cloud

The term cloud describes a group of servers, that are accessed over the internet and the software and databases that runs on those servers [https://www.cloudflare.com/learning/cloud/what-is-the-cloud/]. These servers are located in one or multiple datacenters. There are three types of clouds: Private clouds, which refers to servers and services which are only available internally (i.e. only shared within the organization). The second type are public clouds, which refers to publicly available services (i.e. shared with other organizations) [https://www.cloudflare.com/learning/cloud/what-is-a-private-cloud/]. And lastly, there are hybrid clouds, which mix both of the previous types. All of these have five main attributes in common: They allow for on-demand allocation, self-service interfaces, migration between hosts, as well as replication and scaling of services [lecture notes, VSYS, during bachelor, and https://azure.microsoft.com/en-us/overview/what-is-a-private-cloud/].
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 [Gartner https://www.gartner.com/en/newsroom/press-releases/2021-04-21-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-gro

The current value even exceeds the market capitalization of Norway [https://www.indexmundi.com/facts/indicators/CM.MKT.LCAP.CD/rankings]. Considering the amount of revenue generated (at least $40 billion [https://www.indexmundi.com/facts/indicators/CM.MKT.LCAP.CD/rankings]), it is obvious why the likes as Microsoft (in 2010) and Google (in 2013) followed Amazon into the cloud market [https://www.cbinsights.com/research/amazon-google-microsoft-multi-cloud-strategies/#history].
Cloud computing is able to generate these high rates of revenue because they take advantage of economy of scale, very efficient sharing of resources, as well as a combination of a huge amount of developer effort into a low amount of features (in contrast to every organization implementing the same featureset over and over for themselves) [https://en.wikipedia.org/wiki/Cloud_computing].
Apart from financial and developer efficiency, clouds have a long list of advantages and disadvantages [Domain-specific language for infrastructure as code].
The high degree of automation and possibilities for scaling within a cloud made it possible to scale automatically. The time required to provision (and deprovision) new nodes plays an important role for autoscaling. This is where containers come in.

2.4 Containers

While the idea of containers exists for quite some time already (2006 as cgroups, 2007 with LXC, https://en.wikipedia.org/wiki/Cgroups, https://en.wikipedia.org/wiki/LXC), it only reached mainstream popularity with the release of docker in 2013 [https://en.wikipedia.org/wiki/Docker_(software)]. The main difference between a VM and a container is the kernel: The former has its own dedicated kernel, which runs in parallel with the hypervisors kernel (yet controlled by it). The latter however shares the kernel of the underlying operating system, thus not requiring a kernel to be loaded for each new instance. As a result, the provisioning speed is dramatically reduced: While VMs are not uncommon to exceed 60 seconds until being fully available, containers only require the time the operating system needs to start a new process, which is sub-second in most cases [https://www.vpsbenchmarks.com/labs/provisioning_times].
Containers also (almost completely) solve the „works on my machine“ syndrome, where the developer machine is different to (f.e.) the production system to the extend that a new feature might only work on either, but not both.
Some go even as far as saying containers are the future of cloud computing [https://www.cloudpassage.com/articles/containers-future-cloud-computing/, https://www.devopsonline.co.uk/is-serverless-the-future/, https://www.alibabacloud.com/blog/why-is-serverless-the-future-of-cloud-597191, https://ttpsc.com/en/blog/why-serverless-is-the-future-of-software-and-apps/] (or maybe the future of container computing looks different then previously thought https://azure.microsoft.com/en-us/blog/introducing-the-microsoft-azure-modular-datacenter/, https://patents.google.com/patent/US7278273B1/en).
Docker Inc. also introduced a cross-machine management tool called swarm, which allows users to describe a desired state, which the engine tries to realize (at all

times). It was accompanied by Google's Kubernetes in 2014 on the short list of container orchestrators. Kubernetes is based on another (internal) software by Google called Borg, which is the underlying system for software like YouTube, Gmail, Google Docs and their web search. The company had no place to put the open source software, so they partnered with the Linux Foundation to create the Cloud Native Computing Foundation (CNCF) [https://www.cncf.io/blog/2018/11/05/beginners-guide-cncf-landscape/]. 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 [https://www.alibabacloud.com/blog/why-is-serverless-the-future-of-cloud-computing_597191].
These orchestrators like Swarm and Kubernetes, along with the cloud providers become more complex with the more features they get, and since the high amount of automation leads to an ever-changing state, several ways to describe the desired state were developed.

2.5 Infrastructure-as-Code

IaC takes advantage of multiple factors:

- Software development encompasses more than running it, f.e. a build pipeline, testing and compliance. All of this has to be documented.
- Documentation is hard to hold up to date [How Software Engineers Use Documentation: The State of the Practice, Software Documentation Management Issues and Practices: a Survey]. This is not special to orchestrators or cloud providers, but is true for all software.
- The only source of information that cannot lie (i.e. be out of date) is the sourcecode.
- Scaling (infrastructure) leads to standardized objects.
- In order to have multiple instances of the same type of nodes, they have to be provisioned exactly the same.
- The only (reliable) way to something the same way over and over is to script/program them.
- Infrastructure becomes more and more software defined, reducing required physical changes required for changes in the infrastructure (which enables automation).
- Version-control-systems like git are well established and allow for rollbacks, collaboration, reviews and actionability [Kief Morris Infrastructure as Code]. 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 [Software-defined everything, deloitte, Software-defined everything, researchgate, 2017], the advantages gained by using IaC grow steadily. As soon as a software has an API, it can be integrated into IaC. Since the created code only describes how and when to interact with which API and not the actual implementation behind it, some kind of orchestrator is required which processes the requests and runs the actual workflows behind the endpoints.
There are two ways to implement those workflows. The first is a push-based mechanism, where the orchestrator triggers actions on other parts of the system (f.e. commands a hypervisor to create a VM). The other is a pull-based mechanism, where those subsystems (i.e. a hypervisor) periodically asks the orchestrator whether tasks have to be completed. [`https://www.infoworld.com/article/2609482/data-center-review-puppet-vs-chef-vs-ansible-vs-salt.html`]
These mechanism not only apply to the interaction between the orchestrator and the subsystems, but between the source and the orchestrator as well.
In order to increase the capabilities of the orchestrator or in other words enable more things to get defined via software, middle- or abstraction-layers are introduced. An example for this is the hypervisor that acts as API-gateway between hard- and software-defined machines. The deployment (and configuration) of that middleware (i.e. the hypervisor) is not within the scope of most IaC frameworks and is outsourced. This layer must be as easy to deploy as possible, making it hard to bring in mistakes and staying as flexible as possible for further configuration via software.
It is obvious, that not everything can be software-defined, since some physical objects (like cables) have to be physically placed [Can Infrastructure as Code apply to Bare Metal]. Robots could possibly be used, but in most cases, this is something human workers do. Whether the configuration is correct can often be detected/-measured from software. On the other hand, technologies like FPGAs can even change the CPU architecture via software - so the future might have some surprises in store.
One of the hardest things about applying IaC to bare metal is the complex management and interactions between the multiple APIs. On one side are the „external“ protocols and interfaces like DHCP, TFTP, Hyper Text Transfer Protocol (HTTP), Domain Name System (DNS) and SSH. On the other side are the OSs and the features they provide for automation [Can Infrastructure as Code apply to Bare Metal]. These range from being able to install the OS in an unattended way, over scriptable settings (or better: The non-scriptable ones - looking at you Windows) to compatibility with widespread instance initialization methods like cloud-init [`https://cloudinit.readthedocs.io/en/latest/`].
Another major difference on bare-metal are firmwares. Since they dictate the available version of the hardware APIs, it is important to have them in the correct version [Can Infrastructure as Code apply to Bare Metal].

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 [A Domain Specific Language to Generate Web Applications]. Even among those DSLs the domains they can (and want) describe varies a lot. Additionaly, they differ in several properties, for example whether they are graphical or textual; But since IaC is by definition „as code“, and code is text-based, corresponding DSLs have to be text-based as well. Examples for well-known DSLs in other domains are SQL and CSS `http://www.cse.chalmers.se/~bergert/slides/guest_lecture_DSLs.pdf`. Another property is the approach, which can be imperative or declarative; Imperative languages describe actions to be done, for example „create X additional instances of Y“, whereas declarative languages are used to describe the desired state „I want X instances of Y“. When using the latter, it is the orchestrators job to compare the current state against the described desired state and conclude the required actions themselves [Domain-specific language for infrastructure as code]. Because IaC always aims at describing the whole state, declarative languages are better fitted for this task Infrastructure as Code, Kief Morris. They also have the property of being idempotent: If applied multiple times, the result won't change Infrastructure as Code, Kief Morris. In order to describe the state of infrastructure, the declarative way is also more intuitive. It is the same way humans would describe a state (i.e. „I see three apples“ instead of three times „I see an(other) apple“).
Some DSLs (called „internal“) in this field are based on another language as XML, JSON, or YAML Infrastructure as Code, Kief Morris, second edition. This includes both sub- and supersets of them. Libraries are internal DSLs as well hs script, mode. „External“ DSLs on the other hand are not directly related to other languages lecture notes MODE. An example is the HashiCorp Configuration Language (HCL) used by Terraform Infrastructure as Code, Kief Morris, second edition lecture notes MODE.
An additional difference between the tools and languages is how they are applied. Some use a push-based mechanism, where f.e. the orchestrator initiates communication with nodes and applies changes. Others use a pull-based mechanism, where the nodes need to watch the(ir) configuration at the orchestrator level and execute the required actions locally so they become configured as intended. The design decision of push or pull applies to other things as well: How code changes are loaded into the orchestrator for example.
In contrast to a General-Purpose Language (GPL), a DSL allows better seperation of infrastructure code from other code `http://www.cse.chalmers.se/~bergert/slides/guest_lecture_DSLs.pdf`. Additionally, they are more context driven, which makes them easier to work with for domain experts and users lecture notes MODE. Their syntax is smaller and well-defined too, which makes them less complex as well.
In an ideal world, a DSL for IaC is not a limitation factor; For example it is not limited to neither full usage of virtualization, containers nor bare-metal. It should support

all of those cases and also allow hybrid scenarios. Additionally, it should be able to describe both small and large environments, while the required effort should increase less than linear. Furthermore, an ideal DSL should not lock into a single vendor, but empower migrations and cross-provider scenarios wherever the user sees fit. This includes the licence and owner of the language; It should not be left in the hands of a single organization, but a group (of several organizations/individuals). While a single owning organization tends to reflect itself in the software [`http://www.melconway.com/Home/Conways_Law.html`], a group of organizations or a committee can help in finding a much more universal solution. On the other hand, the more stakeholders are involved, the harder a compromise is to find.].

3 Related work

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

3.1 State-of-the-art automated hardware provisioning

The interest in IaC has been increasing on a steady level over the last years [https://trends.google.com/trends/explore?date=today%205-y&q=%2Fg%2F11c3w4k9rx].
It is estimated that ninety percent of global enterprises will rely on hybrid cloud by 2022 [https://www.idc.com/getdoc.jsp?containerId=prMETA46165020]. 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 [https://www.thestreet.com/investing/public-clouds-are-bright-spot-as-information-technology-spending-eases].
Instead of updating deployed instances, recreating them ensures all of them are equal [Can Infrastructure as Code apply to Bare Metal]. This includes software and firmware upgrades.
Another reason is heterogeneity in systems: Even when using only a single vendor or even a single model, variations occur. Be it that newer models have upgraded firmwares or other „under-the-hand“ changes [Can Infrastructure as Code apply to Bare Metal].
It is better not to assume certain states, but check them instead. This way, whenever a state is unexpected, the automation can exclude this certain node and tell the responsible humans to check what's wrong. The only reliable source of truth for the current state is the current state itself - not some kind of cached or partial version of it [Can Infrastructure as Code apply to Bare Metal].
So far, public cloud providers haven't exactly published how they are provisioning their bare-metal infrastructure.
But there are some hints, as some of those providers have an on-premises or edge product. The Microsoft Azure Stack HCI cluster is such a case. The documentation recommends starters to get machines with the correct drivers and OS preinstalled https://docs.microsoft.com/en-us/azure-stack/hci/deploy/deployment-quickstart. 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 https://docs.microsoft.com/en-us/azure-stack/hci/deploy/operating-system. The preinstalled OS makes the vendor (in this case Microsoft) reponsible for provisioning. So it doesn't solve the problem but shifts it somehwere else. Additionally, it doesn't work in all cases - for example on reinstallations.

While Amazon Web Services Outposts is a similar product, it doesn't allow customers to manage it themselves. Instead Amazon dispatch their own service personnel for every necessary manual task https://aws.amazon.com/outposts/ https://aws.amazon.com/outposts/faqs/.
Google doesn't have a product to bring its whole cloud on-premise or to the edge, but only dedicated featuresets like the Google Kubernetes Engine. The company relies on an underlying VMware vSphere environment and therefore outsources hardware management https://cloud.google.com/anthos/clusters/docs/on-prem/1.3/overview.
When a company like Google relies on a third-party software it has to be special in some way.

So how does deployment of VMwares vSphere clusters work? The most important thing with vSphere is that it can be deployed as a VM on an ESXi server (since version 7.0 the appliance is the only way - previously a Windows system could be used as well), the hypervisor OS developed by VMware https://blogs.vmware.com/vsphere/2017/08/farewell-vcenter-server-windows.html https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.esxi.install.doc/GUID-B64AA6D3-40A1-4E3E-B03C-94AD2E95C9F5.html https://docs.vmware.com/en/VMware-vSphere/6.0/com.vmware.vsphere.install.doc/GUID-ACCD2814-0F0A-4786-96C0-8C9BB5html. In other words, vSphere requires (at least one) manually installed ESXi server, which can then host the vSphere Server software, which then in turn has a feature called Auto Deploy https://docs.vmware.com/en/VMware-vSphere/6.0/com.vmware.vsphere.install.doc/GUID-D0A72192-ED00-4A5D-970F-E44B1ED586C7.html. This feature creates a PXE boot infrastructure that requires an external DHCP server https://docs.vmware.com/en/VMware-vSphere/6.0/com.vmware.vsphere.install.doc/GUID-9A827220-177E-40DE-99A0-E1EB62A4940html https://docs.vmware.com/en/VMware-vSphere/6.0/com.vmware.vsphere.install.doc/GUID-8C221180-8B56-4E07html. The latter has to be configured to distribute network boot details which point to the preexisting vSphere Server https://docs.vmware.com/en/VMware-vSphere/6.0/com.vmware.vsphere.install.doc/GUID-9A827220-177E-40DE-99A0-E1EB62A49408.html. In order to reduce deployment time, Auto Deploy does not install the ESXi OS on machines, but loads the boot image directly into its memory https://docs.vmware.com/en/VMware-vSphere/6.0/com.vmware.vsphere.install.doc/GUID-71F8AE6C-FF4A-419B-93B7-1D318D4CB771.html. This implies that server restarts are equal to redeployments.
Even VMware doesn't seem to bring a TFTP server as part of their software, but describes how to install and configure a third party product themselves https://docs.vmware.com/en/VMware-vSphere/6.0/com.vmware.vsphere.install.doc/GUID-F9056360-544A-4452-8C76-B29018235CEhtml. Since a PXE environment consists of at least a DHCP- and TFTP-server (for performance reasons mostly paired with an HTTP-server) as well as the operating system image, the ratio of reliance on third party products is suprising - considering VMware develops most of their software themselves. But when VMware does rely on this deployment approach, it must have proven to be reliable and hold water.
Apaches CloudStack supports two hypervisors; For ESXi it recommends also using vSphere, while for XenServer and KVM it does not specify any deployment options - it documentation starts after the hypervisor is installed http://docs.cloudstack.apache.org/en/latest/installguide/configuration.html#adding-a-host http://docs.cloudstack.apache.org/en/latest/installguide/configuration.html#adding-a-host-vsphere http://docs.cloudstack.apache.org/en/latest/installguide/configuration.html#adding-a-host-xenserver-kvm.
One of the most recently published cluster software for bare metal is Googles An-

thos. The software and its documentation completely omit the provisioning part up the point where nodes can only be added when they are already accessable via SSH https://cloud.google.com/anthos/clusters/docs/bare-metal/1.6/quickstart.
Common asset management tools like servicenow or i-doit, use providers like vSphere or public clouds for instantiation https://docs.servicenow.com/bundle/rome-it-operations-management-page/product/cloud-management-v2-setup/concept/cloud-mgt-vmware-setup-guide.html or don't support hardware provisioning https://kb.i-doit.com/display/en/VM+Provisioning.
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 https://maas.io/how-it-works https://theforeman.org/introduction.html https://wiki.fogproject.org/wiki/index.php?title=Introduction#What_is_FOG https://fai-project.org/fai-guide/#_a_id_work_a_how_does_fai_work https://cobbler.readthedocs.io/en/latest/index.html https://docs.openstack.org/ironic/latest/ https://rackn.com/rebar/#osp-media https://docs.tinkerbell.org/architecture/ https://docs.microsoft.com/en-us/system-center/vmm/hyper-v-bare-metal?view=sc-vmm-2019. Most often, they have the required software (the aforementioned DHCP- and TFTP-server) embedded in some way and only require minor interactions to configure it properly (like setting up the DHCP range).
Only the minority of bare-metal provisioning software uses or at least supports IPMI as tool of the trade. This includes Canonical MAAS (only for power management), OpenStacks Ironic (for power management and sensor data), RackNs Digital Rebar and ispsystems DCImanager https://maas.io/docs/snap/3.0/ui/power-management https://docs.openstack.org/ironic/latest/ https://rackn.com/rebar/#hw-oob https://docs.ispsystem.com/dcimanager-admin/modules/server-auto-add-module. The main problem with using IPMI for provisioning is its vendor-specific implementations. Not only is it not available for all hardware, but different vendors support different features of IPMI - often even with different APIs. A second, but closely related issue is its unavailability for VMs: Most hypervisors don't support IPMI interfaces for virtual machines. And even if they do (for example via plugins), their documentation is sparse and their development stale https://docs.openstack.org/virtualbmc/latest/.
Another reason for no using IPMI is its historically low security. Although most vendors had their own credentials for accessing the management interface, they used the same combination of user and password for all of their devices https://github.com/netbiosX/Default-Credentials/blob/master/IPMI-Default-Password-List.mdown. With the taking effect of senate bill 327 chapter 886 (1798.91.04) in january 2020, the vendors must now use a unique random password for each machine.
On the other hand, the BMC has capabilities beyond network boot and WOL. For example it allows administrators to debug an unresponsive machine, execute hard resets and change BIOS/UEFI settings remotely. So while IPMI definitely has its own place, it is not the go-to technology for automated provisioning.
Whenever PXE is used for deployments, as a first step an iPXE image is deployed via TFTP. iPXE is best compared to a customizable and very advanced BIOS/EFI but has several advantages over the default ones. For one, it is scriptable https://ipxe.org/scripting. Therefore it is very flexible in its configuration even during its runtime. And it supports loading the actual OS image via HTTP instead of TFTP. Since iPXE is several times smaller and more lightweight than most operating systems,

as well as the fact that HTTP is more performant than TFTP and there exist better tools around it, this approach does not only speed up the deployment but makes it more reliable and customizable, too https://ipxe.org/appnote/uefihttp https://kb.2pintsoftware.com/help/why-is-ipxe-better-that-good-old-plain-vanilla-pxe https://projects.theforeman.org/projects/foreman/wiki/Fetch_boot_files_via_http_instead_of_TFTP https://jpmens.net/2011/07/18/network-booting-machines-over-http/.

The previous part of this chapter focused on the technological „infrastructure" aspect of IaC. Neither less important, less complex nor less diverse is the „as code" part.

As long as there are few properties that change, it is absolutely feasable to use command-line arguments to describe the desired state for IaC tools infrastructure as code, oreilly. But with a growing amount of properties the statespace increases, requiring a better way to describe it: Configuration files. The languages used within those files are mostly DSLs. In contrast to a GPL (not to be confused with the licence), its domain-specific counterpart promises higher success rates even with less experience and significantly higher closeness of mapping Comparing General-Purpose and Domain-Specific Languages - An Empirical Study. 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 Iac, oreilly, kief morris.

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.

3.2.1 Amazon CloudFormation

CloudFormation supports both JSON and YAML, is declarative and typed https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-whatis-concepts.html. 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` https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-whatis-concepts.html.

Instead of requiring a commandline-tool (there is one https://docs.aws.amazon.com/cloudformation-cli/latest/userguide/what-is-cloudformation-cli.html), CloudFormation is designed to work by just uploading the file containing the definition - possible sources are s3-buckets, git-repositories or manual uploads. This implies that the orchestrator is run closed-source by Amazon. Therefore CloudFormation is not only a language by Amazon, but also exclusively for Amazon. Additionally, the user has

no (direct) influence on the capabilities of the language and the orchestrator.
Nevertheless, AWS holds by far the largest market share of the cloud market https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/ and was the first public cloud provider. CloudFormation is therefore one of the earliest DSLs for describing infrastructure. It is widely used https://stackshare.io/aws-cloudformation, and the language itself as well as the tools around it can be assumed to be very mature. There are plugins for most IDEs https://github.com/aws-cloudformation/cfn-lint. 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 https://github.com/aws-cloudformation/cfn-lint. Under the hood, the linter uses schema validation. Assuming the schema is as mature as the language, it can be reasoned that this guarantees validity of the definition files. The linter also provides detailed information on what exactly is wrong in such a file as well, making it quite error-prone. There is a so-called „AWS CloudFormation Designer", too https://console.aws.amazon.com/cloudformation/designer. It aims at giving the user a GUI to create his infrastructure definition files.
As do most languages for IaC, CloudFormation supports custom functions, too. It does so in both JSON and YAML. Examples are `{ "Fn::GetAtt" : [ "logicalNameOfResource", "attributeName" ] }` in the former language and `Fn::GetAtt: [ logicalNameOfResource, attributeName ]` or the short-version `!GetAtt logicalNameOfResource.attributeName` in the latter.
Using CloudFormation to describe infrastructure requires a lot of knowledge: Starting from all the products and features AWS has to offer, over different solutions that have (partially) redundant features, up to understanding the AWS jargon. An example for this is „EC2": New users have a hard time understanding that „EC2" is actually a VM and that there is no „EC1" or similar.
Since CloudFormation is limited to AWS, it is incompatible with bare metal. This also means that it can be ruled out for further usage in this thesis. Nevertheless, it is a big player in the league of DSLs, so examining it for reference does definitely make sense (to some extend).

3.2.2 OpenStack Heat

The Heat component from OpenStack is responsible for IaC. It is not a language by itself, but supports actually two languages. One is named Heat Orchestration Template (HOT) and the other is Amazon CloudFormation. The former is strongly influenced by CloudFormation https://docs.openstack.org/heat/rocky/template_guide/index.html. When the API for CloudFormation is used, the Heat component translates the AWS-specific types to OpenStack compatible ones https://docs.openstack.org/heat/latest/developing_guides/architecture.html.
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 https://docs.openstack.org/heat/latest/developing_guides/architecture.html. The contextual jargon (f.e. „stack") is also inspired by CloudFormation.
Another similarity is about the required knowledge about products/components.

Sticking to the earlier example of creating a VM, new users are required to know that the necessary type is „OS::NOVA::Server“. The „NOVA“-part comes from the fact that the compute component of OpenStack is named this way.
There are three ways to communicate with the Heat component; First, the CloudFormation cli-tool. Then an additional commandline tool for HOT https://docs.openstack.org/mitaka/cli-reference/heat.html and an official library in/for python https://docs.openstack.org/python-heatclient/latest/.
OpenStack supports bare metal via a component called „ironic“ https://docs.openstack.org/ironic/latest/user/architecture.html. It is the closest implementation of what this thesis desires to accomplish https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/15/html/bare_metal_provisioning/sect-introduction. 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 [https://trends.google.com/trends/explore?q=%2Fg%2F11c3w4k9rx]. 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 https://registry.terraform.io/browse/providers. Of those providers „only“ 160 are aimed at IaC https://registry.terraform.io/browse/providers?category=infrastructure. Terraform uses HCL as DSL and is highly plugin-based https://www.terraform.io/ https://www.terraform.io/docs/extend/index.html. In addition to the custom language, JavaScript Object Notation (JSON) is supported as well https://www.terraform.io/docs/language/syntax/configuration.html.
Working with Terraform happens with a commandline executable. The binary loads plugins as needed, communicates with the APIs of the necessary providers and gives feedback to the user. In order to be as efficient as possible, Terraform maintains a local state file. The file contains the last obtained state of the infrastructure. Since Terraform assumes it is the only component changing the infrastructure, this approach enables it to detect differences locally. Afterwards, it automatically generates an execution plan on how to eliminate those differences and reach the desired state. The last step is then the execution itself, which is at the same time the only step where communication with external APIs happens.
Cross-plugin dependencies are supported by the tool as well. This makes Terraform extremely versatile and easily extensible.
Since the state file is meant to mirror the current state, manual interactions with the infrastructure as well as with the state file are strongly recommended against https://www.terraform.io/docs/cli/state/recover.html. Another reason is the difficulty of recovering from such state disasters https://www.terraform.io/docs/cli/state/recover.html.

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“ https://github.com/pulumi/pulumi/tree/v0.3 https://github.com/pulumi/pulumi. Instead of using Yet Another Markup Language (YAML), JSON or HCL, Pulumi is available as library in several programming languages. Available are these libraries for Node.js, JavaScript, TypeScript, Python, Go(lang), and .NET Core (therefore C#, F#, and Visual Basic).

3.2.5 Open Cloud Computing Interface

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

Two non-vendor-specific standards for describing IaC in a formal way have emerged. First, Open Cloud Computing Interface (OCCI) which was published by the Open Grid Forum (OGF) Open Grid Forum in 2011 https://www.ogf.org/documents/GFD.183.pdf. Their organizational member list mirrors their mainly academic purpose https://www.ogf.org/ogf/doku.php/members/organizational_members. 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 https://occi-wg.org/about/index.html. 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) https://occi-wg.org/about/index.html. OCCI is developed by the Open Grid Forum, which is backed by companies such as Dell EMC, NetApp and Oracle www.hpcinthecloud.com/hpccloud/2007-09-17/whats_next_for_ogf.html https://www.networkworld.com/article/2332351/consortium-formed-to-promote-enterprise-grids.html. Since the launch of the standard, several open source cloud providers have started to support it, including OpenStack, OpenNebula and CloudStack https://occi-wg.org/community/implementations/index.html. The last update of the specification was in 2016 https://occi-wg.org/index.html. The IT worlds changes fast, so five years since the last update are a long time.

3.2.6 OASIS TOSCA with Simple-Profile

The second cross-vendor standard that has emerged is called Topology and Orchestration Specification for Cloud Applications (TOSCA). It was first published in 2013 by the Organization for the Advancement of Structured Information Standards (OASIS). The latter is also well-known for widespread standards like Advanced Message Queuing Protocol (AMQP), Message Queuing Telemetry Transport (MQTT), OpenDocument, PKCS#11, Security Assertion Markup Language (SAML), SARIF and VirtIO https://www.oasis-open.org/standards/. 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 https://www.oasis-open.org/committees/membership.php?wg_abbrev=tosca,https://www.oasis-open.org/committees/tosca/obligation.php https://www.oasis-open.org/member-roster/. The latest contribution was only one week before the time of writing, so its actively pursued and developed http://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html https://www.oasis-open.org/committees/documents.php?wg_abbrev=tosca.
TOSCA has been used in some proof-of-concept projects [Domain-specific language for infrastructure as code] in 2019, but their results were disappointing: The interfaces between the core standard and the supported providers are described as to be always out of date making even simple operations impossible. The tools of the ecosystem surrounding the standard are said to be non-user-friendly and their learning curves to be flat [https://www.admin-magazin.de/Das-Heft/2018/02/Apache-ARIA-TOSCA]. Still, TOSCA has a lot of plug-ins for platforms like OpenStack, VMWare, AWS, Google Compute Platform (GCP) and Microsoft Azure (Azure), configuration management tools like ansible, chef, puppet and saltstack or container orchestrators like docker swarm and kubernetes [https://www.admin-magazin.de/Das-Heft/2018/02/Apache-ARIA-TOSCA, https://docs.vmware.com/en/VMware-Telco-Cloud-Automation/1.9/com.vmware.tca.userguide/GUID-43644485-9AAE-410E-89D2-3C4A56228794.html]. All those projects conclude that the standard is extremely promising, but the current state makes it impossible to use properly [https://www.admin-magazin.de/Das-Heft/2018/02/Apache-ARIA-TOSCA].
Since then, TOSCA 2.0 was released, which introduced huge changes like the transition from XML- to the current YAML-declaration.
The standard contains the specificiation of a file archive format called Cloud Service ARchive (CSAR). These archives contain five major parts https://www.opentosca.org/documents/howto-build-csars.pdf:

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

In addition to the TOSCA standard itself, OASIS also published an extending standard called „TOSCA Simple Profile“ https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html. While large parts of both specifications are redundant, Simple Profile provides the types and ecosystem needed for real-world applications of TOSCA. The reference implementation of the TOSCA orchestrator, OpenTOSCA interprets and executes whatever is necessary of a CSAR definition is also compatible with the TOSCA Simple Profile https://www.opentosca.org/documents/howto-build-csars.pdf.

3.2.7 OASIS TOSCA with Cloudify

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

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 `https://docs.ansible.com/ansible/latest/collections/community/general/hpilo_boot_module.html` `https://docs.ansible.com/ansible/latest/collections/dellemc/openmanage/idrac_network_module.html`. But for „general“ bare-metal provisioning it mostly relies on external systems like cobbler `https://docs.ansible.com/ansible/latest/collections/community/general/cobbler_system_module.html#stq=netboot&stp=1`.

3.2.9 Others

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

3.3 Culling 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 Infrastructure as code - Final Report, John Klein Infrastructure as Code, Kief Morris https://www.terraform.io/intro/vs/cloudformation.html https://www.terraform.io/intro/vs/chef-puppet.html https://www.atlassian.com/continuous-delivery/principles/infrastructure-as-code.

These two categories are named differently in different sources, f.e. using „Infrastructure Definition“ as synonym for provisioning and „Configuration Registry“ for configuration management. Some software like Ansible can also fill both roles Infrastructure as Code, Kief Morris.

The field of configuration management is mainly platform-agnostic. Or more specifically, it does not matter whether it is applying to cloud instances, VMs or bare-metal. Most of the tools in this area interact with a preexisting API for their tasks. This could be the API of a cloud provider or SSH access to a VM or bare-metal machine.

In order to bootstrap a whole infrastructure with not preexisting APIs except for the ones available at a hardware level, this thesis focuses on DSLs that are aimed at provisioning. At the same time, it is relatively easy to create instances (of whatever) by sending requests to a cloud provider or a hypervisor, while doing the same with bare-metal not so much: There is no such API - yet.

As a result, all configuration management DSLs are not eligible. Namely, these are Ansible, Chef and Puppet (from the introduced languages at least).

As described earlier, all languages that are imperatively describing infrastructure can be ruled out as well. Funnily enough, this doesn't apply to any of the introduced DSLs for IaC.

As already stated earlier, Amazon CloudFormation is too AWS-specific to be easily extended to bare-metal and is therefore also not an option.

The implementations of OCCI-compatible software are severely outdated, and their interrelated deprecation/maintenance levels are confusing to say the least https://occi-wg.org/2012/07/18/occi-in-openstack/index.html https://occi-wg.org/2012/07/18/occi-in-opennebula/index.html https://github.com/tmetsch/occi-os https://github.com/stackforge/occi-os. That, and because the latest release of the OCCI standard was over five years ago, as well as sources stating OCCI software does not work even with basic examples, together with an accompanying recommendation to use TOSCA instead, OCCI will not be looked at in too much detail as well. For example, the documentation for the OpenStack implementation https://occi-wg.org/2012/07/18/occi-in-openstack/index.html recommends to visit the corresponding wiki-site, which is even older https://wiki.openstack.org/wiki/Occi.

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 https://docs.openstack.org/heat/latest/. Additionally, there are few examples, and to run it even in a proof-of-concept style requires to install and run many components of OpenStack. These requirements make it not only unfeasable for smaller infrastructures, but for from-scratch”deployments like the one desired in this thesis as well.

TOSCA has several implementations and corresponding (inofficial) extensions like

Cloudify, Alien 4 Cloud or Puccini. It is ouf of this thesis' capabilities to compare all of them. Therefore, only the official extension, named „Simple-Profile" will be included in the comparison.

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

Language	State/Description
Ansible, Chef, Puppet	Ruled out because they are made for configuration management, n
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.
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 Comparative Study of DSL Tools Comparing General-Purpose and Domain-Specific Languages - An Empirical Study Domain-specific language for Infrastructure-as-Code [Stachowiak, Allgemeine Modelltheorie]. They compare based on attributes like (no specific order):

- Primary approach Comparative Study of DSL Tools
- Guarantees provided in case of well-formedness Comparative Study of DSL Tools
- Reusability of components Comparative Study of DSL Tools
- Error proneness and reporting like line number and column offset Comparative Study of DSL Tools Comparing General-Purpose and Domain-Specific Languages - An Empirical Study
- Efficiency: Amount of code for a given case study Comparative Study of DSL Tools
- Aspects to learn for a given case study or how hard the mental operations are Comparative Study of DSL Tools
- Viscosity: How hard it is to make changes/updates Comparing General-Purpose and Domain-Specific Languages - An Empirical Study

- Hidden dependencies like requiring agents, a dedicated server or a third-party software Comparing General-Purpose and Domain-Specific Languages - An Empirical Study
- Visibility: How easy is it to find the responsible snippet in the codebase Comparing General-Purpose and Domain-Specific Languages - An Empirical Study
- Extensibility: Can the language be adapted to environment changes
- Maturity (documentation, user-base, community): How good are edge-cases documented and how well is the product established
- Ecosystem

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

3.5 Selection

Since previously, many DSLs were ruled out, this thesis is going to look at only three languages in more detail. These are HashiCorp Configuration Language (HCL), Pulumi (Golang-SDK) and TOSCA in combination with its „Simple-Profile“ extension.
The following overview in table 3.2 summarizes the results of the comparison. The score in each cell is either „average“ (abbreviated with „~“), „worse“, or „better“. By default every score has the value „average“. Depending on the comparison to the average features, the values can then be „worse“ or „better“.

Tabelle 3.2: *Overview over the DSL comparison results - TODO adjust to new structure*

Dimension	HCL/Terraform	Pulumi	TOSCA Simple-Profile
Approach	~	~	~
Guarantees	~	~	~
Reusability	~	~	~
Errorproneness	~	~	~
Amount of code	~	~	~
Aspects to learn	~	~	~
Viscosity	~	~	~
Dependencies	~	~	~
Visibility	~	~	~
Extensibility	~	~	~
Maturity	~	~	~
Ecosystem	~	~	~

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 1 expresses the very same as the HCL in code-snippet 2

```
1  "resource": {
2    "aws_instance": {
3      "example": {
4        "instance_type": "t2.micro",
5        "ami": "ami-abc123"
6      }
7    }
8  }
```

Listing 3.1: *JSON example*

```
1  resource "aws_instance" "example" {
2    instance_type = "t2.micro"
3    ami = "ami-abc123"
4  }
```

Listing 3.2: *HCL example*

The language is able to display the same amount of information in a denser way. This has both positive and negative effects: On one side, the fewer brackets enable

users to focus on the actual content. On the other side, it is a complete new languages, and developers first need to learn it.
Because the architecture of Terraform is highly plugin-based and these plugins are often developed by third parties, the documentation of the core tool never mentions anything about guarantees.
The plugin system is cure and blessing at the same time. It enables extraordinary extensibility, but makes it sometimes hard to reuse snippets - what works for one provider most often doesn't work for another, too. The system allows for cross-plugin dependencies, so that really every infrastructure integration can be described. But the quality of the ecosystem depends on the quality of the plugins in most cases. Terraforms user-base is huge, and it is the current state-of-the art. Therefore, its maturity is very high, most edge cases are covered, the documentation is great, and the very same applies to the most common plugins.
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 applies to the desired bit as well.

3.5.2 OASIS TOSCA with Simple-Profile

As described earlier, infrastructure in TOSCA is modelled via CSAR files. The main part 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. Because the parts (for example a webapplication on a webserver that references a database on another server) can often be reused for other infrastructure parts (or other applications), it makes sense to split some kind of generic description for those parts from the concrete 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 some kind of generic template of a component. The standard allows for different kinds of types; The most intuitive type is the node type. In TOSCA context, nodes does not refer to only machines or servers, but all components that can be defined via software. This includes low-level applications like operating systems, intermediate middleware programs like web- and database-servers, as well as high-level software like the webapplications running on them. The other types are more specialized. For example artifact types are used to describe constant artifacts like OS-images, scripts and other external (as in non-TOSCA) files. The other type categories are data-, capability-, interface-, relationship-, group- and policy-types. They are mostly self-explaining, but the capability and requirement system of TOSCA is worth additional description. In order to reflect where relationships are allowed (for example a webapplication cannot run on a database server), the standard has the aforementioned system. Corresponding capabilities interlock with requirements like puzzle pieces. An example: Assuming a node has a requirement „container runtime", it can only be assigned to an underlying node that offers a capability with the same name. To satisfy this constraint, TOSCA orchestrators know they need a node that offers that capability. If there isn't one, they will try to create one. If the

container runtime itself has other requirements like „compute“, the orchestrator will resolve those as well.

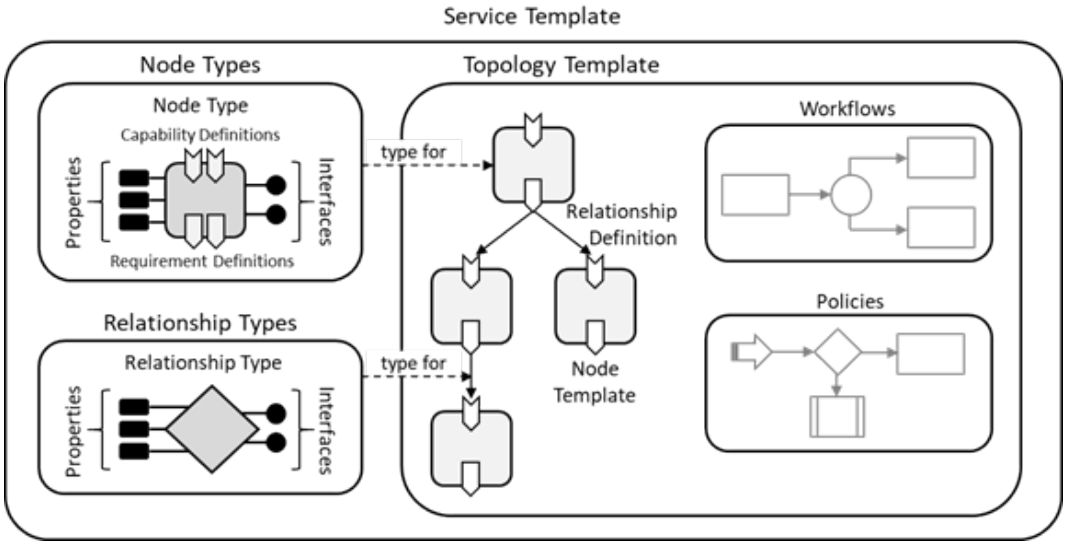

Abbildung 3.1: *Architecture of TOSCA and the components of CSAR files*

The last part contained in CSAR files are the management plans. They describe the lifecycle of nodes and how to achieve them. Examples are instantiation, configuration and deallocating/uninstallation/deletion (depending on the resource type).

The standard supports imports with or without namespacing. One potential import is the Simple-Profile extension. It adds a basic set of such predefined types to the standard, for example the node-types Compute, Webserver, and Webapplication with necessary requirements and capabilities.
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 architecture of the CSAR files and the ability of inclusions and substitutions make it relatively easy to find the corresponding code for a certain component.
This, and the fact that versioning is not only possible but even required by design make changes in the infrastructure or lifecycles easily managable.

3.5.3 ONLY READ UNTIL HERE

3.5.4 Pulumi

...

3.6 Comparison of existing Domain-Specific-Languages

3.7 Everything-as-a-Service

- can openstack do all of this? - stability - legacy code - complexity - Rackspace-as-a-Service; will-on-prem die? (”Why on-prem won’t die”) no, security of data, costs, privacy, pressure/trust, (with or without pdu, usc, ups) - Metal-as-a-Service; vs VM-as-a-Service (vps), noisy neighbour, vSphere AutoDeploy, Ironic, tinkerbell, ipv6, which os, ipmi, kernel/firmware integrity, zones, pdu, psu, rack, sdn - Network-as-a-Service; topology, vlan, sdn, both on hw and sw layer - DNS-as-a-Service; global or not? via k8s? - Hypervisor-as-a-Service; esxi, kvm (used by aws, gcp (no qemu)), node-size, vm-size, compare to metal-as-a-serice (differentiate) - Compute-as-a-Service; vm-as-a-service, vps - Encryption-as-a-Service: ram, disk, network on host/node-level (TPM?) - Storage-as-a-Service; alternative to rook, hyper-converged vs dedicated (SVC by IBM), sds, both on hw and sw layer - IAM-as-a-Service; web-authn with yubikey, cloud-iam, 3rd-party iam like github oauth (openstack iam? ad necessary? why no ad join for nodes? -> linux, ephemereal, cattle) - k8s-as-a-Service; which os, in-memory-os, cluster-api (gardener, how to configure nodes? Terraform, ansible, cloud-init, ignite), why multi-tenacy via multi-cluster? - IaC-as-a-Service; generation/compliance with OPA, CRD-like formal description, check GCP, AWS, Azure and Openstack for common ground - secrets-as-a-Service; turtles all the way down presentation, SCM, orchestration, Secrets-as-a-service (hashicorp vault?) meta/mgmt - bare-metal-marketplace

3.8 Example reference infrastructure

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

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

3.9 Issues with existing standards and frameworks

- no comparison of iac dsls - not enough effort to integrate with other tools / dsls / clouds (vendor-lock-in seems desired) - either no proper standard (vendor-specific) or not enough support for multiple vendors -> everyone reinvents the wheel and wants to establish the own work as industry-standard -

4 Design and Implementation

4.1 Requirements

- Understand basic TOSCA (CSAR files, file-structure, yaml-structure)
- Understand TOSCA-extensions (at least the one for hardware)
- wol capabilitiy
- live-os / hardware-information
- ssh command execution
- feedback to user

4.2 Architecture

- executable, library
- why and which modules/packages
- sub-commands (init, install, uninstall, ...)

4.3 Features

4.3.1 TOSCA package

The first task for the implementation to understand the TOSCA standard in high detail, so it can be properly implemented. This included research on preexisting implementations. Sadly, most of the larger implementations were based on one another, and the most complete of them all was severly outdated. The other implementations were unfinished and could not be taken as references.
The library that was developed is the result of a line-by-line read-through with parallel implementation of the whole TOSCA standard specification.
With it, it is possible to parse any Service Templates, Types, and Topologies. In addition to parsing them and creating Go-native structures out of them, basic functionalities like the functions or validation of elements, it is limited to one implementation type, Bash, while the specification states that Python should be supported as well. There are several occasions, where the specification is very unclear, or simply incomplete. These cases are mostly about edge cases and were not required for the

proof-of-concept this thesis tries to achieve. Therefore, they were left out of the implementation.
Because the basic TOSCA specification describes how to work with extensions like the TOSCA Simple Profile, it was possible to implement such imports, references, and relations as well. This means, the TOSCA library (called package in Golang context) is fully compatible with the TOSCA Simple Profile and all other extensions.

4.3.2 CSAR package

After the TOSCA package was able to parse the contents of file, the last chapter of the specification was implemented as a seperate package. It contains information about how to pack multiple artifacts like OS-images, definition-, or other required files together into one CSAR file. The file is basically a zip-archive, but the contents need to follow a certain schema. For example there are three places where metadata like version and name can be placed. If they are not found there, the whole file is invalid.

4.3.3 Command-execution package

In order to run Bash commands from the application itself and retrieve their outputs, a complete package about command execution was built. It makes it easy to run commands that are defined in TOSCA definitions.

4.3.4 Docker control package

During the implementation phase it was clear very early, that some kind of DHCP- and TFTP-server will be required. And those will require easily repeatable setup. In most cases, this can be solved with docker. But because the goal of this thesis was to bring all required bits together, it was necessary to create docker images, start containers (with parameters like volumes and forwarded ports), as well as stopping and removing the containers when they are not needed any more.
The official docker binary (for linux) is created in Golang as well, and the software is open source. Docker even provides an SDK for other developers to integrate communication with the docker engine into their binary. Sadly, the documentation is sparse and the few examples displayed along the SDK were often not enough to get even seemingly things like container stopping to work. For this particular example it was necessary to add the container stopping and removal twice: One time when the application terminates successfully, the container fulfilled its job and isn't needed any more. And a second time, when the application terminates due to an error somewhere else, and the default termination does not happen. Even „deferring" the container termination did not work. Only when the SIGTERM interrupt was „manually" listended for and a function was implemented to remove

the container in such a case, the removal was successfull in all cases.
Another obstacle was on how to retrieve live logs of the container and embed the retrieved output in the logs of the application. It was solved by creating a buffered streamreader, which was polled periodically for contained linebreaks.

4.3.5 Live-OS image generation

After the application was able to handle docker containers, a replicatable way to create a live-OS was neede. It should create an iso-file from scratch, with no previous requirements (apart from an internet connection). Since tiny linux images should be relatively common, this was first estimated to be an easy task.
As one can image, it turned out differently. The first linux distribution that was tested was „Minimal Linux Live“. tests with different operating systems (size, speed, repeatable, webserver-capable)
live-debian details

4.3.6 DHCP-, TFTP-, HTTP-server package

docker container, ports, variable config vs fixed config

4.3.7 Wake-on-lan package

actual wol vs simulated wol for hypervisors

4.3.8 SSH package

key-generation, variable key path, key placement on servers
actual command execution and feedback returning

4.3.9 TOSCA hardware extension

types, topology, tests

5 Evalution / Analysis / Discussion

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

6 Conclusion

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

- looked at several DSLs, their requirements and compared X ones in more detail
- added additional elements to the comparison
- improvement recommendations for TOSCA standard
- implemented on-demand bare-metal provisioner, an up-to-date tosca-library, which is also hugely extensible.

7 Outlook

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

Abbildungsverzeichnis

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