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

Akronyme

AMQP Advanced Message Queuing Protocol

API Application Programming Interface

AWS Amazon Web Services

Azure Microsoft Azure

BMC Baseboard Management Controller

BOOTP Bootstrap Protocol

CNCF Cloud Native Computing Foundation

DHCP Dynamic Host Configuration Protocol

DSL Domain-Specific Language

GCP Google Compute Platform

IaC Infrastructure-as-Code

IPMI Intelligent Platform Management Interface

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

PXE Preboot eXecution Environment

SAML Security Assertion Markup Language

SSH Secure Shell

TFTP Trivial File Transfer Protocol

TOSCA Topology and Orchestration Specification for Cloud Applications

VM Virtual Machine

WOL Wake On LAN

Table of contents

Akronyme 3

1 Introduction 6

2 Background 8
2.1 Bare-metal . . . 8
2.2 Virtualization . . . 10
2.3 Cloud . . . 11
2.4 Containers . . . 12
2.5 Infrastructure-as-Code . . . 13
2.6 Domain-specific language . . . 14

3 Related work 16
3.1 Current state of the art of Infrastructure-as-Code and related trends 16
3.2 Provisioning tools . . . 16
3.3 Comparison of existing Domain-Specific-Languages . . . 16
3.4 Everything-as-a-Service . . . 18
3.5 Example reference infrastructure . . . 19
3.6 Issues with existing standards and frameworks . . . 20

4 Design and Implementation 21

5 Evalution / Analysis 22

6 Discussion 23

7 Conclusion 24

Liste der Codebeispiele 26

Anhang 27

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 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, Red Hat's OpenShift 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) 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 BIOS/UEFI settings, the machine starts with its configured boot-order, for network-boot this means an embedded Network Bootstrap Program (NBP) (f.e. pxelinux or ipxe), which is like a networking equivalent to what GRUB is for local disks: It downloads a kernel from a network resource, loads it into memory and finally (chain-)boots it [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 makes the NIC 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 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 allowed 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, like DRAC for DELL, ILO for HPE and IMM for IBM. Probably due to their origin an 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. Network-booting on the other side helps with automating a high number of servers in parallel, but doesn't really help with debugging errors.
These standards are to state-of-the-art remote-server-administration-tools for several years, along with Secure Shell (SSH). They mostly solve the administration scaling problem or form the base for other tools.
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, the web-server, it still blocks

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. Microsofts Hyper-V followed in 2008, as well as the Proxmox Virtual Environment. The initial release of VMwares ESX hypervisor 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. 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.

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) to 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. 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]. In order to describe the state of infrastructure, the declarative way is 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 apple“).
In contrast to general purpose languages, DSLs allow better seperation of infrastructure code from other code. Additionally, they are more context driven, which makes them easier to work with for experts and users. 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 Current state of the art of Infrastructure-as-Code and related trends

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].

3.2 Provisioning tools

tools/frameworks - openstack ironic (https://docs.openstack.org/ironic/latest/user/architecture.html) - installs OS on a local disk -> should the OS be installed on a local disk or every boot happen via network? -> depends on boot-count and network speed and desired first-boot-time and second-boot-time - verify-HW; verify a node is accessible with HPMI (could be combined with flashing nodes' firmware) - ...

- default passwords for IPMI/BMC were vendor-specific (calvin...) but due to new US law they must be random: senate bill no 327, chapter 886 - add title 1.87.26 to part 4 of division 3 of civil code, relating to information privacy

3.3 Comparison of existing Domain-Specific-Languages

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 to support multiple providers. It uses HCL https://www.terraform.io/ as DSL and is highly plugin-based [https://registry.terraform.io/browse/providers].

[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/dok
Yet, the website of the OCCI standard reveals that the last contribution happened back in 2016, so this project seems to be abandoned since then (at least neglected).
Second, the Topology and Orchestration Specification for Cloud Applications (TOSCA) standard was first published in 2013 by the Organization for the Advancement of Structured Information Standards (OASIS). The latter is also responsible for well-known standards like Advanced Message Queuing Protocol (AMQP), Message Queuing Telemetry Transport (MQTT), OpenDocument, PKCS#11, Security Assertion Markup Language (SAML) and VirtIO so its name is well-known in the world of software. Additionally, its members are not only an overwhelming number of academic or governmental institutions but een more so global players like Cisco, Dell, Google, Huawei, HP, IBM, ISO/IEC, SAP and VMware [`https://www.oasis-open.org/committees/membership.php?wg_abbrev=tosca`,`https://www.oasis-open.org/committees/tosca/obligation.php`]. The latest contribution was only one week before the time of writing, so its actively pursued and developed [`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 said 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 flat / all but steep [`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`].
While AWS CloudFormation only works for a single provider, being developed by the same company that provides the infrastructure its determined to manage is a major advantage. cloudformation was the first?
OpenStack Heat `https://www.slideshare.net/openstackil/heat-tosca` HOT == Heat Orchestration Template, YAML only came to replace cloud formation syntax following the cloudformation limited model hot is only for infrastructure creation tosca is application centric by design -> tosca is more universal hot workflow hardcoded in heat engine toscas interfaces allow for any workflow -> no hardcoded workflow tosca to hot translator project developed by ibm, huawei and others -> goal is to describe stack in tosca and use heat cloudify uses tosca templates directly soon to use heat to orchestrate infrastructure adds monitoring, log collection, analytics,

workflows

tosca adopted hot input and output parameters, which took that from cloudformation hot added software_config provider to describe application stack explicitely hot adopted tosca relationship syntax and semantics

https://www.oasis-open.org/committees/download.php/56826/OpenStack%202015%20Tokyo%20Summit%20-%20TOSCA-and-Heat-Translator-TechTalk.pdf TOSCA-Parser"by IBM, can parse TOSCA Simple Profile in YAML "Heat-Translator", maps and translates non-heat (f.e. tosca) templates to hot supports tosca csar Murano-= OpenStack's application catalog that provides application packaging, deployment and lifecycle management - plans to integrate tosca csar

https://wiki.openstack.org/wiki/Heat/DSL2 evolve first heat/dsl and incorporate tosca and CAMP

Terraform - AWS CloudFormation - https://en.wikipedia.org/wiki/RAML_(software) -> supported by aws api OpenStack Heat -> can use TOSCA Cloudify -> uses TOSCA ... (see notes)

Originally, TOSCA was meant to work only with XML, but since some year or version it also supports YAML.

terraform is very similar to tosca, but because its usability is higher and its learning curve is steeper, its a lot more user friendly.

- CAMP http://docs.oasis-open.org/camp/camp-spec/v1.1/cs01/camp-spec-v1.1-cs01.pdf , https://en.wikipedia.org/wiki/Cloud_Application_Management_for_Platforms - terraform (describes itself as standard: https://www.terraform.io/intro/vs/custom.html) - cloudformation https://www.terraform.io/intro/vs/cloudformation.html

3.4 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; webauthn 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

3.5 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

3.6 Issues with existing standards and frameworks

- no comparison of iac dsls - not enough effort to integrate with other tools / dsls / clouds - 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

- 5-10 pages - goal: fellow student understands content and would be able to more or less reproduce work - legitimate chosen approach - develop own ideas, trace them to existing theories - analysis and development - why was the approach (algorithm/technique/...) chosen and how does it work - show how concepts from theory are applied - test setup and achieved results

possible steps: - requirement study - analysis / design -> UML, interaction, behavioural model, basic algorithms/methods, detailed description of models and their interactions (class/sequence diagrams) - manual, how to use program/device - system development and implementation

—

considerations: - ipv6 not 100 percent necessary, but would be good - easy to learn - easy to adapt (to counterpart-interface updates) -> plug-in system? - tosca-orchestrator: - 4.3ff of `http://docs.oasis-open.org/tosca/TOSCA-Instance-Model/v1.0/csd01/TOSCA-Instance-Model-v1.0-csd01.html#_Toc500843787` ”orchestrators manage the state of nodes and transitions them from state to state. This notion of state is somewhat artificial in that the orchestrator assumes a stable state is reached after an operation executes [...] without erroran error results in an undefined state”(no automatic rollback defined in tosca) ”orchestration states are only valid during orchestration. [...] the orchestrator or the imperative workflow [...] must decide the current state of all nodes in the topology. ”[...] event stream can be maintained for the life of a deployment [...]As nodes are transitioned through their states, a subset of attributes and relationships may be defined. [...] This requires that in general TOSCA implies semantics such that not all attributes would be available in a given state. Nodes are only visible when they have a state defined (i.e. the orchestrator is dealing with their lifecycle)node attributes are only defined for the stable statesnode relationships are always navigable when the source and target node exists[...] state is never updated outside an orchestration. [...] no way to propagate state changes from the node to the orchestrator and nodes don’t have a state attribute.> no state file! Nodes can update their attributes with no specific guarantees in terms of precision or accuracy”

5 Evalution / Analysis

- 5-10 pages - case studies - how does it work in close to real-world settings - how well does it work (scale, speed, stability)

6 Discussion

- 5-10 pages - introduction to discussion - why where the results as they were? what was expected? - new insights - limitations - recommendation for future research

7 Conclusion

- 2-4 pages - do NOT summarize the thesis -> thats what the abstract is for. (but you can summarize most important results as introduction) - synthesize findings and conclude what can be learnt from them - refer to research questions and answer them (confirm, rejected). can be sections - clarify whether result/software meets requirements as stated earlier. - compare with results/software from others - finish with an outlook on how work could be continued - ideas - better technique - unsolved problems

Abbildungsverzeichnis

Liste der Codebeispiele

Anhang

Anhang A

Ein erster Anhang

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