Master Thesis

zur Erlangung des Grades „Master of Science“

Leveraging open-source infrastructure-as-code for enterprise environments

an der	Technischen Hochschule Ulm Fakultät Informatik Studiengang Intelligent Systems
vorgelegt von	Till Hoffmann
Matrikelnummer	3135572
für die	Daimler TSS GmbH
Betreuer	Benjamin Gotzes
Erstgutachter	Prof. Dr. rer.nat. Stefan Traub
Zweitgutachter	Prof. Dr.-Ing. Philipp Graf

Eingereicht am 31.10.2021

1 Demo-chapter

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut efficitur maximus diam, aliquam ultricies tortor semper non. Sed lobortis dolor vel facilisis auctor. In metus ante, venenatis at accumsan non, ultricies in ante. Maecenas placerat felis ipsum, nec luctus ligula pellentesque eu. Nulla dapibus neque cursus augue varius, in ultricies dolor malesuada. In bibendum imperdiet nisi, ut feugiat neque pellentesque eget. Nulla tortor mauris, sodales id arcu quis, aliquet pulvinar est. Praesent lobortis faucibus magna quis vestibulum.

1.1 Demo-sub-chapter

Vestibulum rhoncus eget mi ac accumsan. Praesent metus nisi, pellentesque ut nunc sit amet, eleifend mattis tellus. Nullam rhoncus tellus nec augue laoreet, quis cursus nisl dictum. Quisque scelerisque nisl et volutpat hendrerit. Nunc cursus nibh dolor, venenatis pulvinar dui lacinia sit amet. Aliquam diam sapien, consectetur sodales tempor non, eleifend a sem. Pellentesque quam tortor, placerat a enim vitae, porta laoreet neque.

1.2 Second-demo-sub-chapter

Sidenotes for the second demo-sub-chapter

Suspendisse lectus lacus, eleifend et velit ut, luctus elementum nisl. Nunc rhoncus ultricies metus, in feugiat odio tempor nec. Nulla rutrum, urna eu suscipit ornare, nisl metus venenatis neque, ut consectetur nulla urna vitae mauris. Aliquam ac neque ut velit ultrices mattis. Vestibulum sodales vulputate arcu quis congue. Maecenas nec odio tempus, fringilla nisl laoreet, condimentum felis. Quisque auctor quam vel augue blandit vehicula.

Higlighting, ~~striking through~~, underlining and several more simple formatting opportunities are possible.

Emphasizing text is also possible.

A long hyphen — can be written as well.

Insert links like this: `https://github.com/thetillhoff/master-thesis`

Acronyms are written down as: Yet Another Markup Language (YAML). Following occurences are then displayed as: YAML.
Entries of the glossary are written down as: YAML. Following occurences are then displayed as: YAML.

In this line you can see an example for `inline code`.

Here follows a complete codeblock with caption and red highlighting:

```
# Folgende unterschiedliche Einrückung derselben Ebene ist in YAML erlaubt:
firstlevel1:
  secondlevel1: somevalue
  secondlevel2: anothervalue
firstlevel2:
    secondlevel3: alsoanothervalue
    secondlevel4: andonemore
# Folgende unterschiedliche Einrückung derselben Ebene ist in YAML nicht
↪   erlaubt:
firstlevel3:
  secondlevel5: hello
    secondlevel6: world
    secondlevel7: exclamationmark
```

Codebeispiel 1.1: *YAML Einrückungen*

and one without caption and highlighting:

```
"group":["John Smith","Jane Smith"]
```

Use
\newline for newlines and

\clearpage for pagebreaks
(\cleardoublepage is used to also skip the next page as well if it is odd numbered and the document is set to twosided)

Different vertical spaces are:
smallskip,

medskip and

bigskip

Text can be on the left side \hfilland on the right side.

\vfill makes text to appear at the bottom of a page.

An unordered list:

- First item
- Second item
- Three dots either as ... or as ...

An unordered list with specific seperation space:

- First item
- Second item

An ordered list, where the numbers start after 3:

4. Some
5. example
6. content

Now, an image follows - also with caption:

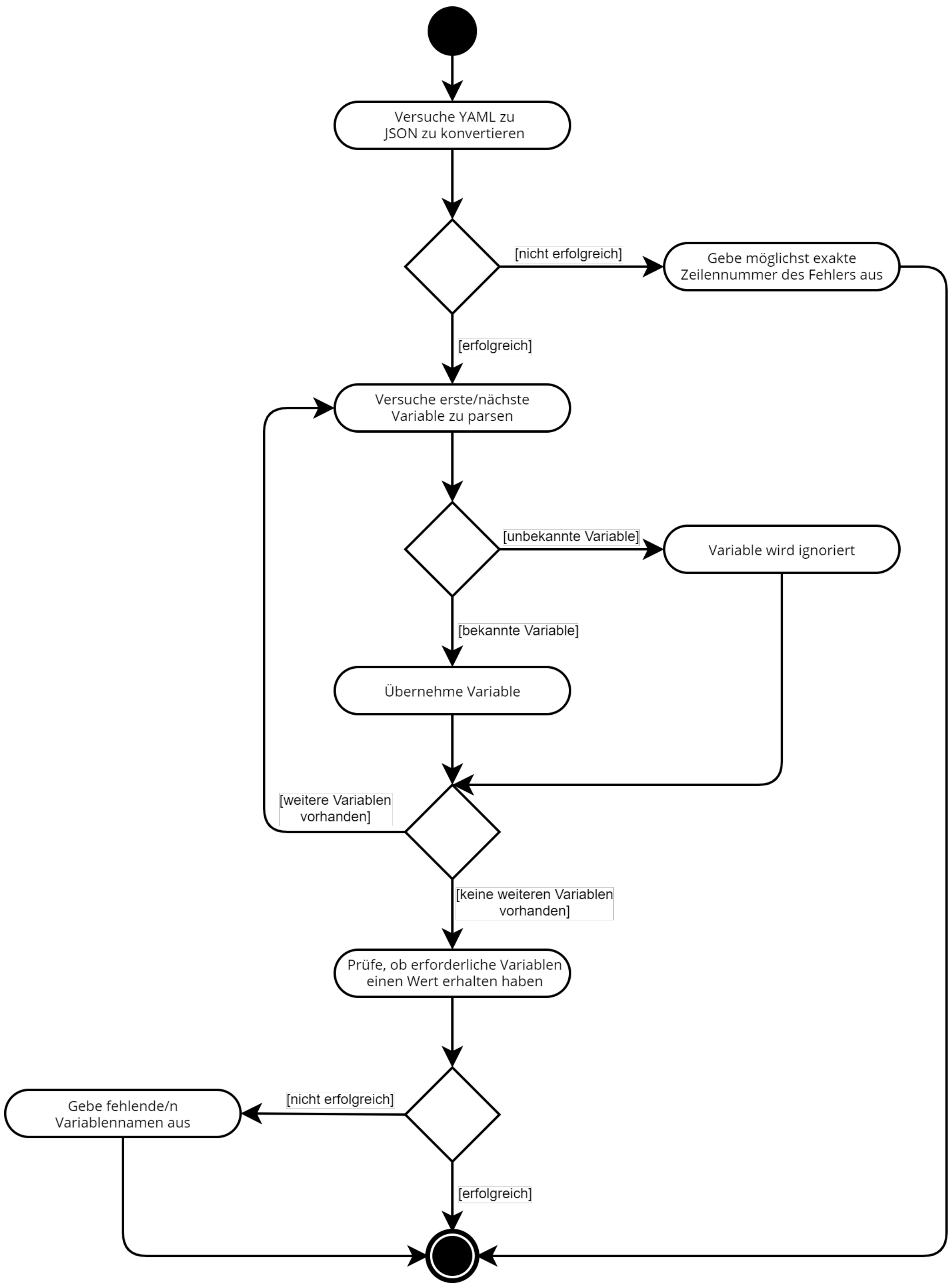
Versuche YAML zu JSON zu konvertieren
[nicht erfolgreich]
Gebe möglichst exakte Zeilennummer des Fehlers aus
[erfolgreich]
Versuche erste/nächste Variable zu parsen
[unbekannte Variable]
Variable wird ignoriert
[bekannte Variable]
Übernehme Variable
[weitere Variablen vorhanden]
[keine weiteren Variablen vorhanden]
Prüfe, ob erforderliche Variablen einen Wert erhalten haben
[nicht erfolgreich]
Gebe fehlende/n Variablennamen aus
[erfolgreich]

Abbildung 1.1: *UML-Aktivitätsdiagramm: Verarbeitung von an Helm und Kubernetes übergebenen Variablen*

Abstract

Bla/laber/fasel

Glossar

YAML	YAML is yet another markup language.

Akronyme

AMQP	Advanced Message Queuing Protocol
AWS	Amazon Web Services
Azure	Microsoft Azure
BMC	Baseboard Management Controller
BOOTP	Bootstrap Protocol
DHCP	Dynamic Host Configuration Protocol
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
YAML	Yet Another Markup Language

Inhaltsverzeichnis

2 Introduction

- 3-4 pages - why is the thesis relevant, what is the problem/task - context - scientific importance / motivation / relevance - goal, at least one research question - not too technical - introduce to structure of thesis (which chapters are to be expected)

—

motivation == vision … it started with a vision from star treck …

3 Background

- 7-22 pages - research on history, ”story”on the topic - introduce necessary science/engineering to understand approach. - decide on what is commonly known and what not. ->

—

Searching online for Infrastructure-as-Code (IaC) quickly leads to the terms such as „snowflake“, „pet“ and „cattle“. In this context, the former two are synonyms and refer to directly/manually managed (configured and maintained) machines. The latter is used when referring to machines, which are never directly interacted with; All administrative interactions with them are automated. This culture of treating machines as cattle aims to solve the administration effort for large amounts of servers. When operating on such a scale, it is easier to maintain some kind of automation framework and unify the deployment of machines than to administrate each server manually. 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 worked whenever necessary to understand the further chapters and point out where the pain-points of the then-state-of-the-art tooling were.

3.1 Bare-metal

In the early times of datacenters, they required quite the administrative effort. Alone for reinstalling an operating system, a server would require one administrator physically located close to the server, some kind of physical 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. But these devices only scale so far, since they mostly could only connect to a handful of servers. 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 prices are insane nowadays, considered they basically only need to switch some connections, but that is out of scope here.

To 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). 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 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", it triggers the machine to power-on. Depending on the BIOS/UEFI settings, the machine might start Network Bootstrap Program (NBP), which is like a networking equivalent to what GRUB is for local disks: It loads a kernel into memory before chain-booting it [`https://docs.openstack.org/ironic/latest/user/architecture.html`]. The combination of all those technologies 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 (ignoring the relative minimal amount of storage TFTP requires).

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 press buttons. And additionally, while it is possible to network-boot the installation media, the installation itself (more specifically: going through the installation wizard) has yet to happen manually as well.

The next generation of servers (since 1998) had such a remote control integrated into their mainboard, rendering KVM obsolete, because this new method outsources the scaling issuescales vertical: For every new server, that embedded chip adds its corresponding necessary features as every new server comes with its own 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 there are those globally respected 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-pc-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 were to state-of-the-art remote-server-administration-tools for several years, along with Secure Shell (SSH).

In the meantime, operating systems evolved as well. They added features like cloud-init (2008, https://github.com/canonical/cloud-init/releases?after=ubuntu-0.3.1)

With the physical scaling problem now resolved, another pain-point emerged: When a larger set of hardware needs to be installed, one administrator per installation is required. This means either a lot of administrators are needed, or slower installation time since hardware gets processed in a sequential way.

3.2 Virtualization

Even though IBM first 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 making sense in terms of performance, consolidation leading to lower management overhead, lesser 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-source-killer-app] 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, where in contrast bare-metal nodes required a downtime for for every hardware change like RAM upgrades and hardware replacement due to failures. The same feature also drastically improves disaster recovery capabilities [https://searchservervirtualization.techtarget.com/definition/server-virtualization].

But the use of hypervisor and clustering them for live-migration and other cross-node functionalities has downsides as well: Vendor lock-in, since the different Virtual Machine (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 expertise for the additional software.

Yet, 100 percent of the fortune 500 and 92 percent of all business used virtualization technologies in 2019 [https://www.statista.com/statistics/1139931/adoption-virtualization-technologi https://www.vmware.com/files/pdf/VMware-Corporate-Brochure-BR-EN.pdf]. On a sidenote, VMWare claims that 80 percent of all virtualized workloads run on VMWare technology [https://www.vmware.com/files/pdf/VMware-Corporate-Brochure-BR-EN.pdf], whereas Statista estimates their share to only 20 percent [https://www.statista.com/statistics/915091/global-server-share-physical-virtual/].

Either way those numbers are impressive and highlight the importance of virtualization.

- converged infrastructure: combine storage and normalnetwork traffic into the same network -> no fiber channel switches any more - simpler and easier to manage, easier and cheaper to purchase - hyper-converged infrastructure: merge storage nodes and compute nodes; or in other words: every node has storage and a powerful processor. - hope for less over- and underuse of resources, easier to scale - no additional appliances for data protection, data de-duplication (integrated in HCI software) - performance guarantee, predictable at all times - disaggregated hyper-converged infrastructure / hybrid hyper-converged infrastructure - composable infrastructure - ...

3.3 Cloud

Public clouds have quite the arsenal of pros and cons [Domain-specific language for infrastructure as code]. - vps - https://www.vpsbenchmarks.com/labs/provisioning_times vps provisioning times - hetzner: 14s - GCP: 20s - Amazon EC2: 23s - DigitalOcean: 24s - Scaleway: 32s - Alibaba: 41s - Amazon Lightsail: 48s - Oracle Cloud: 60s - Vultr: 62s - OVHcloud: 72s - IBM Cloud: 86s - MS Azure: 102s

3.4 Containers

- container - kubernetes

- myths: - virt has significantly lower perf than bare-metal - bare-metal container is better than container in vm (f.e. lower overhead) -> no live-migration on bare-metal, chunks of bare-metal (sizing)

3.5 Infrastructure-as-Code

There are two types of IaC; push, where a central software tells all other components about their desired state and pull, where all components query a central software about their desired state.

3.6 Domain-specific language

There are two types of domain-specific languages; declarative, where the final state is described and imperative, where operations are described [Domain-specific language for infrastructure as code].

3.7 Scope

limits of IaC (like physical cable) -> software defined (datacenter/*) FPGA config-level - vms or not should be able to be answered on config-level and therefore by user not by architecture of iac - poc: on-demand/self-service k8s-clusters for users - cli-wrapper around lib, no webgui (as simple as possible) - api for everything <-> everything as a service, all levels (bare-metal, virt, ...) - implies full automation - optional: transparent costs mgmt/meta: - cost-limit per user/group/departments/...

goals - no vendor-lock-in - scalable (startup, dc -> 1node to 1000 node) - declarative ? - open-source

4 Related work

- 7-22 pages - core element of the thesis, as it shows I'm able to write scientifically. - explain what other researchers have found on the topic. - existing approaches (current state of the art, theories, literatur (seperated by subtopic)) - other scientific contributions to solve the task - explain gap in literature, that the thesis is trying to fill. - new method - new data - new application

4.1 Current state of the art of Infrastructure-as-Code and related trends

google trends: related topics, worldwide: - AWS - Terraform - DevOps - Jenkins - cloud computing - azure - automation - Ansible -> in usa: - cicd - yaml - plugin - compliance - pipeline -> 5y old worldwide: - git - agile software development - policy - hashicorp - serverless — similar queries: - terraform - aws

- k8s - multi-tenancy

4.2 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) - ...

- Why DSL and not general purpose language? - smaller and well-defined / less complex - seperates infrastructure code from other code - domain-specific, so easier to work with for experts and customers

- 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

4.3 Comparison of existing Domain-Specific-Languages

DSL == modeling language [A Domain Specific Language to Generate Web Applications]

——— ———————————————————— "'yaml CloudFormation: approach: - declarative - JSON and YAML - typed (AWS::ProductIdentifier::ResourceType, f.e. AWS::EC2::SecurityGroup) - how does the typing system work: field typefor all components - create "stack update"(execution path), save it in s3-buck or anywhere, then run it. no direct api access? hidden-dependencies: - AWS only structure: - mappings: - ... - metadata: ... - outputs: - ... - parameters: - <variablename>: - Default: ... - Description: ... - Type: ... - minlength, maxlength, regex pattern for strings, ... - Constraintdescription - resources: - WebServer: - type: AWS::EC2::Instance - properties: - UserData: ... - KeyName: Ref: <variablename> validation / error reporting: - custom schema validation (?) - only checks if parent language is well-formatted (json,yaml) - additional tools - python cfn-lint as plugin for IDEs or static analysis tool - ruby cfn-nag for static analysis, aimed at security - python taskcat for making a testrun with a template -> integration tests - dry-run possible aspecty-to-learn: - Fn::..."in JSON, "!..."in YAML -> custom functions like Join, GetAtt - JSON or YAML is required, the graphical designer does only the rawest of work alternatively: use the AWS Cloud Development Kit (AWS CDK) for Typescript, python, java or .net - AWS products, their naming and how each components attributes are named tooling/ecosystem: - integrated in aws-cli - "designer": https://eu-central-1.console.aws.amazon.com/cloudformation/designer/home?region=eu-central-1 - any text-editor for JSON/YAML, or use the AWS Cloud Development Kit (AWS CDK) for Typescript, python, java or .net - git pre-commit validation with kfn_nag"(https://github.com/stelligent/cfn_nag) and taskcat"(https://github.com/aws-quickstart/taskcat) optimizations: - parallel resource creation/updating/deletion, can be controlled with "DependsOn"attribute extensibility: - https://aws.amazon.com/cloudformation/fea
There is a CloudFormation Registry, where third-party sources can provide a template for their app guarantees: - cost calculator: https://calculator.aws/ - no guarantee updates work (f.e. insufficient permissions, account quota limit); automatic roll-back on failure reusability of components: - there is the cloudformation registry where apps can be published - Refcan reference to other components - nesting of "stacks"is possible Visibility: - predefined structure, splitting into different files possible Viscosity: - updates of the DSL are made by aws - updates to your apps have to be made manually - updates have to be validated manually Consistency: - (probably) pretty high, as developed by one team and being an official product by amazon. notes: - fails if dependend component is neither defined nor already existing - state-monitoring: events; but those are not saying much (only time, type of the event origin (alarm, stack,...), logical ID (custom name), physical ID (instance id), status (f.e. CREATE_COMPLETE, CREATE_IN_PROGRESS), Reason (User initiated)) - costs: - AWS::*, Alexa::* and Custom::* are free - 3rd party providers cost stuff - Naming resources restricts the reusability of templates and results in naming conflict when an update causes a resource to be replaces"[https://aws.amazon.com/cloudformation/faqs/] -> naming is not possible

for all components — Heat: approach: - declarative - compatible to AWS Cloud-Formation - OpenStack-native REST API, CloudFormation-compatible Query-API - either cli or api directly - YAML - python app - heat cli accesses heat-api, which sends API-requests via RPC to heat-engine, which does the heavy lifting - python-heatclient lib (== api) hidden-dependencies: structure: - parameters image_type: type: string label: Image Type description: Type of instance (flavor) to be used default: m1.small constraints: - allowed_values: [m1.medium, m1.large, m1.xlarge] description: Value must be one of m1.medium, m1.large or m1.xlarge. - resources my_instance: type: OS::Nova::Server properties: key_name: … image: … flavor: get_param: image_type - outputs instance_ip: description: The IP address of the deployed instance value: get_addr: [my_instance, first_address] validation/error-reporting: - heat template-validate: used to validate a template, with inserting templated values and ignoring specific errors -> feels like dry-run aspects to learn: - YAML - HOT structure, types, functions and how each components attributes are named tooling/ecosystem: - cli - python-lib optimizations: extensibility: - there is NO app catalog (`http://lists.openstack.org/pipermail/openstack-operators/2017-July/013965.html`) for heat templates guarantees: reusability of components: - works with cloudformation stuff, and is closely related to it visibility: - see cloudformation viscosity: - see cloudformation (?) consistency: - similar to cloudformation, but with openstack types notes: - imitates AWS Cloudformation on several occasions (types, structures, wording (f.e. stack)) - there is a openstack REST-API AND a CFN-API - cli-command names are confusing - ”heatmakes the clouds rise — Terraform: approach: hidden-dependencies: structure: validation/error-reporting: aspects to learn: tooling/ecosystem: optimizations: extensibility: guarantees: requsability of components: visibility: viscosity: consistency: notes: — Tosca/cloudify: approach: hidden-dependencies: structure: validation/error-reporting: aspects to learn: tooling/ecosystem: optimizations: extensibility: guarantees: requsability of components: visibility: viscosity: consistency: notes: — Tosca/simple-profile: approach: hidden-dependencies: structure: validation/error-reporting: aspects to learn: tooling/ecosystem: optimizations: extensibility: guarantees: requsability of components: visibility: viscosity: consistency: notes: “‘ ——

Languages to compare: - CloudFormation - Heat - Terraform - TOSCA/cloudify - TOSCA/Simple-Profile

model: formal representation of entities and relationships [Stachowiak, Allgemeine Modelltheorie] - abstraction: don’t desribe all attributes, only the relevant ones - isomophism: statements for model entities hold for the real world - pragmatism: model is build to fulfill a purpose for set of users, during certain time, for certain activities

modeling language is defined by: - abstract syntax -> which elements exist in a model and how can they be combined - concrete syntax -> how are the model elements and their relations expressed - semantics -> what is the meaning of each model element

approach: - model-based (original goal is documentation, communication and analysis, model is secondary artefact) - model-driven (model is used to generate code or being interpreted for extracting actions, model is primary artefact) - increased development speed via automated transformations - defined output, repeatable process - seperation of concerns using views - reuse of transformations

aspects of mdsd workflow: - always: - define modeling language (abstract and concrete syntax) - tool support to define models - persist models in files or dbs - tranform models to text - often: - transform model to model - transform text to model - analyze and interpret models - automate workflows from single steps

domain expert: - uses modeling language to create models technology experts: - creates modeling languages and transformations

external dsl: - dedicated toolkit to create modeling language internal dsl: - customize existing programming or modeling language (usually textual) - idea: define an api in an ordinary programming language such that accessing this API looks like using specialized dsl

metamodel: model of modeling language, defines - concepts that can be used to build model - their attributes - how they can be combined - their meaning (usually not though)

why meta models: allows to reuse technology for - storing models in files - storing models in dbs - creating a form-based editor - creating a graphical editor - generating code from models - transforming models into other models metamodel is used to describe abstract syntax, not concrete syntax!

MOF Meta Object Facility: domain specific modeling language to describe metamodels contains - classes (meta-class) - properties/attributes (meta-...) - associations (meta-...) - generalisation (meta-...) - packages (meta-package) There is a meta-metamodell: mof described with mof original - model - metamodel - meta-metamodel

types of concrete syntax: - diagram - text (programming language) - tables - trees and forms - XML - custom - combinations

why textual languages are popular for computer science: - easy to edit (cut, copy, paste) - easy to create parsers and reuse existing SE tools - accessible with console based tools - single input type; can be worked on with only using the keyboard - compare and merge - version control - no special tools needed (only simple text editor) - can be supported by IDEs bad: - complex dependencies hard to recognize

approaches: - parsing: - concrete syntax primary artifact - concrete syntax (text) is directly edited and converted (parsed) to an AST (model) - projection - abstract syntax primary artifact - AST (model) is rendered to concrete syntax (text), edit actions directly on AST (model) (AST=Abstract Syntax Tree)

what to compare: - abstract syntax: meta-model which defines parts/domain-concepts/model-elements and rules/validation of model - concrete syntax: representation of model (instances) f.e. in an editor - static semantics: evaluatable without executing/interpreting the model - dynamic semantics: what the model means or expresses what to compare them on: - kinds of DSLs: textual DSL, visual DSL, library in a programming language, configuration tool, wizard - textual - internal: library in an existing programming language - exernal: seperate language - visual - configuration tool - wizard - approach, f.e. templating, translation or term rewriting - declarative, imperative - executability, hidden dependencies - optimizations; automatic hinting, execution optimization - extensibility -> quality - efficiency/lines-of-code -> quality - tooling (developing AND using) -> ecosystem - guarantees - reusability of components - error reporting -> error-proneness, progressive evaluation - aspects to learn / learning curve - viscosity; how hard is it to update stuff - visibility; how hard is it to find a certain part, is it clear where relations are - consistency

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

Two non-vendor-specifc 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/dol
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, Amazon Web Services (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-Automa
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

4.4 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

4.5 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

4.6 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 -

5 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?

6 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)

7 Discussion

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

8 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