Orchestrating Future Service Chains in the Next Generation of Clouds

Roman-Valentyn Tkachuk* Dragos Ilie* Kurt Tutschku*
*Blekinge Institute of Technology (BTH)
[roman-valentyn.tkachuk|dragos.ilie|kurt.tutschku@bth.se]

***Abstract*—Service Chains have developed into an important concept in service provisioning in today's and future Clouds. Cloud systems, *e.g.,* Amazon Web Services (AWS), permit the implementation and deployment of new applications, services and service chains rapidly and flexibly. They employ the idea of Infrastructure as Code (IaC), which is the process of managing and provisioning computing infrastructure and its configuration through machine-processable definition files.**

In this paper, we first detail future service chains with particular focus on Network Function Virtualization (NFV) and machine learning in AI. Afterwards, we analyze and summarize the capabilities of today's IaC tools for orchestrating Cloud infrastructures and service chains. We compare the functionality of the major five IaC tools: Puppet, Chef, SaltStack, Ansible, and Terraform. In addition, we demonstrate how to analyze the functional capabilities of one of the tools. Finally, we give an outlook on future research issues on using IaC tools across multiple operators, data center domains, and different stockholders that collaborate on service chains.

I. Introduction

Service Chains have developed into an important concept in service provisioning in today's and future Clouds. Cloud systems, *e. g.,* Amazon Web Services (AWS) [1], permit the implementation and deployment of new applications, services and service chains rapidly and flexibly. They employ the idea of Infrastructure as Code (IaC) [2], which is the process of managing and provisioning computing infrastructure and its configuration through machine-processable definition files. Network Function Virtualization (NFV) uses the concept of network service chaining [3], which is the capability of applying Software-Defined Networking (SDN) mechanisms to create a virtual service chain of connected network services, *e. g.,* a chain containing a firewall, a NAT, and an Intrusion Detection System (IDS). This Network Service Chain (NSC) is subsequently provisioned to a specific customer. The idea of service chains is also increasingly applied in data-driven AI system engineering. Data-driven AI applies Machine Learning (ML) frameworks, *e. g.,* TensorFlow, that turn the engineering or inference process into one that uses chains of AI services and AI functions chains. The chains are denoted as AI pipelines [4], [5], [6]. In security testbeds, service chains can be used to instantiate test scenarios, consisting of network, hardware, software, attackers, and vulnerable entities.

The provisioning of the above-described service chains can be done manually. However, this approach might be a highly time consuming and error-prone process. Using the IaC idea, however, may accelerate the provisioning of the above-outlined service chains. IaC can cut the time to deploy a new service chain from hours to minutes. In addition, it may easily provide for automatic scaling of the service chain, either by allowing rapidly instantiating parallel service chains or permitting very large chains comprising hundreds or even thousands of service providing entities.

In Clouds, IaC obtains these features by treating the infrastructure as a software system, and by applying software engineering practices to manage changes to the system in a repeatable, structured and safe way.

The paper is structured as following: Section II gives a brief overview on what kind of service chains are addressed in this paper, their specific needs, the standards and data structure to model and describe them, and on examples of tools which might be considered for orchestration and how these tools are related to tools for IaC. Section III details the test environment for testing the IaC tools. Section IV provides an in depth comparison of IaC tool, already with an outlook to future service chains. Section V describe the experiments to validate the functional capabilities of a specific tool (Ansible) and describes our plans for future work.

II. Service Chains

Service chains have become a major concept for service deployment in SDN and NFV environments [7]. NSCs or Service Function Chains (SFCs) are chains of connected network services, such as firewall, network address translation, and intrusion detection systems [8]. These chains form again a service that can be provisioned. Operators may set up a catalog of service functions that can be embedded in a chain. A provisioned network connection can use one or multiple services functions with different characteristics. Figure 1 depicts two concurrent NSCs provisioned to different customers. The chains are implemented on top of a NFV platform and use four different network functions. NFV-based service chains are typically instantiated within a NFV cloud platform of a single operator. The service function may be implemented in these clouds through virtual machines (VMs) or by light-weight containers, which are executed on demand.

Service chains, however, are not limited to SDN and NFV. Recent machine learning frameworks, such as TensorFlow [9], [4], also apply sequenced functions denoted as *AI pipelines.* Figure 2 shows an AI pipeline for keyword spotting in spoken

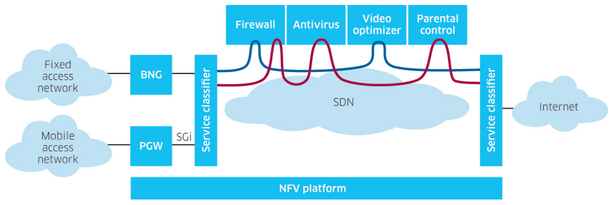

Fig. 1. Example of NSC implemented by an NFV platform, cf. [8]

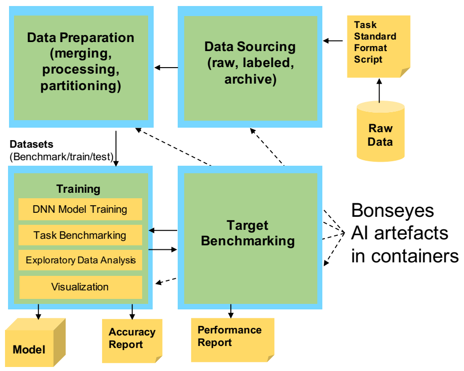
Data Preparation (merging, processing, partitioning)
Data Sourcing (raw, labeled, archive)
Task Standard Format Script
Raw Data
Datasets (Benchmark/train/test)
Training
DNN Model Training
Task Benchmarking
Exploratory Data Analysis
Visualization
Target Benchmarking
Bonseyes AI artefacts in containers
Model
Accuracy Report
Performance Report

Fig. 2. Bonseyes pipeline for keyword recognition, cf. [5]

sentences. The pipeline is used in the H2020 project Bonseyes as a use-case for demonstrating the efficiency of collaborative development in data-driven AI systems [10], [5]. The pipeline shows four AI functions: *Data Sourcing*, *Data Preparation*, *Training*, and *Target Benchmarking*. These functions are denoted in the AI engineering process of Bonseyes as *AI artifacts* and are implemented as Docker containers. An AI pipeline is a chain of AI artifacts. A similar AI pipeline for AI inference, denoted as InferLine [6], is depicted in Figure 3. While the InferLine pipeline focuses on single physical execution location, *e. g.*, a single data center, the Bonseyes pipeline is designed considering a distributed execution environment of AI artifacts. Bonseyes assumes that the AI pipeline is executed in a federated IT infrastructure, which is initially not secure and where hosts are a-priori not trusted. Hence, the use of federated distributed execution requires specific and new concepts for data security, Digital Rights Management and data privacy [11].

Service chains can be modelled as a special class of graphs.

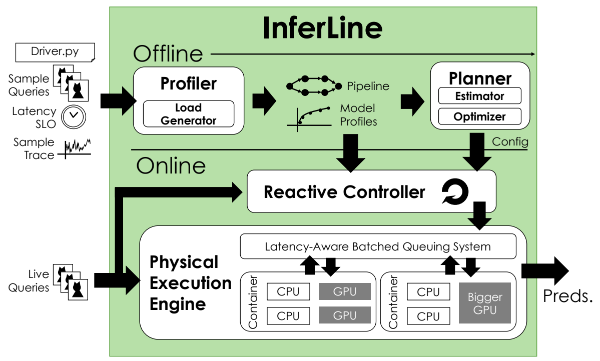

Fig. 3. Inferline Pipeline, cf. [6]

In NFV these graphs are denoted as *Virtual Network Function - Forwarding Graphs (VNF-FGs)*. The ETSI Open Source MANO project [12] provides an information model and data structures for VNF-FGs [13]. The Tacker project [14] is an OpenStack service for NFV orchestration with a general purpose VNF Manager to deploy and operate Virtual Network Functions (VNFs) and Network Services on an NFV Platform. Tacker together with IaC tools, such as Ansible (see below), enabled highly scalable service chain deployment and orchestration in OpenStack [15].

III. Test Infrastructure

At Blekinge Institute of Technology (BTH) we are currently designing and implementing a federated service chain infrastructure, shown in Fig. 4, that aims to address the following objectives. First, it should enable convenient and efficient testing during the implementation of a service chain. In particular, it should be easy to integrate the infrastructure in a continuous integration / continuous testing / continuous delivery (CI/CT/CD) environment for Agile software development and DevOps. Secondly, it should aid in diagnosing issues with existing service chains. For example, an existing service chain (or a scaled-down version) would be replicated on our infrastructure. The infrastructure would be instrumented to enable debugging of API calls between service elements and to provide various levels of introspection into the actual elements. The replica can then be configured to enter the state that triggers the issue to be diagnosed. Finally, we would like to be able to scale up the test resources by allowing interested parties to interconnect infrastructure under their control to our infrastructure.

The first two objectives aforementioned require the provisioning of servers that execute the service chain and the proper configuration of the chain elements. By provisioning, we mean the ability to allocate servers to the chain and to install the required software. Proper configuration implies that the elements have the necessary information to interconnect and interoperate such that the intended chain functionality is implemented. Although both provisioning and configuration can be done manually, such a process is error prone and time consuming. In addition, in the case of CI/CT/CD, one often wants to begin with a system in a pristine condition and therefore will frequently provision and reconfigure. Consequently, it is desirable to automate these operations, which is why we consider IaC.

The last objective, interconnecting infrastructure from multiple parties, typically results in a federated architecture, where parties retain control over their infrastructure and can control how much resources they share with the federation members.

We have decided to use OpenStack [16], a cloud operating system, for implementing the test infrastructure. This choice was motivated by elements in a service chain often being implemented as VMs or light-weight containers inside a cloud, by our desire to provide elastic resource allocation, and by our past experience in operating an OpenStack node [17]. In

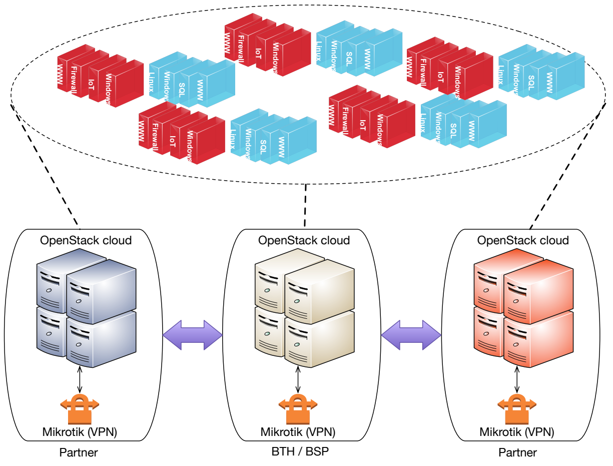

Fig. 4. Federated test infrastructure

addition, OpenStack can support a federated architecture [18], which is consistent with our third objective.

The design of the test infrastructure is currently driven by two specific use-cases. The first use-case, which is ongoing work, is about deploying the Bonseyes AI pipeline described in Section II. The elements of the pipeline will refuse to interoperate unless mutual trust is established with neighboring elements in the sequence and with the host they execute on. The potential presence of malicious entities poses a number of interesting threats [11] that such a system must handle in order to enter a trustworthy state [19]. Our intention is to use the test infrastructure to emulate such malicious entities and verify that the system can handle the threats. The second use-case is about scanning IPv6 networks to discover active hosts. The scanning can be useful, for example, for penetration testing exercises. However, the tremendous size of IPv6 networks poses a serious challenge for classic scanning algorithms that iterate sequentially over the entire address space. We have developed a set of IPv6 scanning heuristics [20] to overcome this challenge by trading accuracy for execution time. We are using the test infrastructure to deploy virtual IPv6 networks and evaluate the performance of the heuristics. Although IPv6 networks do not represent typical service chains, their use allowed us to analyze the functionality of a IaC tool as described in Section V of the paper.

IV. IaC Tools Comparison

Infrastructure as Code (IaC) is the process of managing computational resources through machine-readable definition files, instead of making manual configuration changes or using interactive configuration tools [2].

We have performed a qualitative comparison of five IaC tools, as shown in Table I. The aim was to evaluate their suitability for the testbed described in Section III, according to a set of desirable features extracted from web sites hosting the software, online descriptions and comparisons [21], [22], as well as own practical testing.

The features were divided into four main categories: architectural concepts, usability, security, and popularity. In Table I the categories are separated by double horizontal lines.

A. Architectural concepts

The **type** of a IaC tool denotes its main purpose. An *orchestration* tool is used foremost for provisioning nodes for a specific role (*e. g.*, web server). Typically, this means installing a base image (containing operating system and software) on bare metal. *Configuration management* tools assume there is a base image installed already on the server. Their use is mostly to install and manage software on existing machines. However, none of the compared tools is a purely configuration management or orchestration tool and the label just indicates where their major functionality lies.

Tools with *mutable* **infrastructure** allow software to be installed or updated incrementally on each node, independently from other nodes. With more and more updates, nodes will have diverging update histories. This can lead to a phenomenon known as configuration drift [22], where each node becomes different from other nodes with similar functionality. *Immutable* infrastructure implies that the infrastructure is stateless because the state of the cloud environment is specified as a whole, and thus will not face any configuration drifts.

In a *server-only* **architecture**, IaC software is installed only on the server, whereas in a *client-server* architecture there is also agent software installed on the clients. Typically, the agent requires that extra ports are opened in the firewalls to enable communication with the server software. Thus, the administration effort and the attack surface increases in the case of client-server architectures.

B. Usability

The availability of the **source code** can have a strong impact on community interest, code base knowledge and contributions to IaC tool development. All IaC tools included in this study are using *open source* licenses, which in most cases grant free use and distribution of software modifications.

The **changes propagation method** feature indicates how the changes are delivered to the target machines. In a *push* approach, the IaC server keeps track of what installed on each client and applies changes where necessary. In the *pull* approach, clients request updates from the IaC server. Typically, this is a more scalable approach than push, because the clients maintain on their own the state of installed software.

The **programming language** feature indicates the scripting language used to write configuration scripts. This may be important if the IaC tool is integrated in a larger ecosystem where a specific programming language is required. However, the **programming language approach** can have a larger impact on the type of changes that can be automatically deployed. With a *declarative* language, one expresses the end state of the infrastructure, without specifying each and every state in the configuration script. The disadvantage is that it is not possible to fully control the flow of script execution. Contrary, *imperative* languages allow full control of execution flow in

TABLE I
IaC Tools Comparison

	Puppet	**Chef**	**SaltStack**	**Ansible**	**Terraform**
Type	Configuration Management	Configuration Management	Configuration Management	Configuration Management	Orchestration
Infrastructure	Mutable	Mutable	Mutable	Mutable	Immutable
Architecture	Client-Server	Client-Server	Client-Server	Server-Only	Server-Only
Source Code	Open Source	Open Source	Open Source	Open Source	Open Source
Changes propagation method	Pull	Pull	Pull	Push	Push
Programming Language	Puppet DSL	Ruby DSL	YAML	YAML	HashiCorp Configuration Language
Programming Language Approach	Declarative	Imperative	Declarative and Imperative	Declarative and Imperative	Declarative
Interoperability	Master - Unix, Agents - Unix/Windows	Server - Unix, Clients - Unix/Windows	Master - Unix, minions - Unix/Windows	Server - Unix, supports configuration of Windows	Vast majority of operating systems
Accountability	Activity service API	Ruby based Chef Log Resource class	SaltStack built-in Logs	Ansible built-in Logs	Terraform built-in Logs
Authentication	Certificate-based	Certificate-based	Publisher ACL and External Auth Modules	Cloud built-in API	Cloud built-in API
Availability	Multi-master architecture. If Primary master fails, Secondary master takes its place.	Contains primary server as well as backup server.	Multi-master architecture. If Primary master fails, Secondary master takes its place.	Contains primary instance which is substituted by secondary instance.	Contains single active node that is substituted by secondary node.
Integrity	Periodic client configuration validation	Periodic client configuration validation	N/A	N/A	N/A
Confidentiality	HTTPS	HTTPS	SSH	SSH	Cloud build-in API
Contributions (GitHub)	432	477	1596	1488	653
Libraries	4435	3052	240	8044	40
StackOverflow Threads	2639	4187	614	3633	131
Year of First Release	2005	2009	2011	2012	2014

the configuration code. The disadvantage of this approach is that the code tends to have a higher failure rate and that it requires a deeper knowledge of the programming language. Also, this type of code has a low rate of re-usability.

Finally, the **interoperability** feature indicates the compatibility of a tool with different operating systems: Unix/Linux, Windows, MacOS, OpenBSD.

C. *Security*

The **accountability** feature describes the ability of IaC tool to log (*e. g.*, through `syslog`) details about configuration changes including some form of identity (*e. g.*, IP address, user name) for the entity that triggered the changes. It is of course desirable to allow only specific entities to trigger changes. This is captured by the **authentication** feature which indicates the ability of IaC tools to authenticate user/software that attempts to use the IaC tool. This is particularly important in a federated testbed, because federation members may want to implement policies concerning who can propagate changes to their hosts.

The robustness of the IaC tool against different types of failures and crashes is captured in the **availability** feature.

The **integrity** feature indicates the ability of IaC tool clients to detect or undo configuration changes coming from outside the IaC tool (*e. g.*, the local admin).

Traffic generated by IaC tools may contain confidential information (*e. g.*, authentication tokens) and should therefore be encrypted. This is captured by the **confidentiality** feature.

D. *Popularity*

The features in this category attempt to describe the level of popularity of the IaC tools, for example in terms of code **contributions** on GitHub or the number of **StackOverflow discussion threads**.

The **Library** feature indicates how many optional extensions/plug-ins are available for each tool. However, it should be noted that in many cases the libraries have overlapping functionality.

Finally, the **Year of First Release** provides some course indication about the maturity of the tool in question.

V. Evaluation and future work

Based on the comparison from Section IV, we decided to evaluate Ansible as the IaC tool for the test infrastructure described in Section III. The reasons for our choice is that Ansible is well integrated with OpenStack through the *ansible-openstack* project. This brings broad cloud orchestration capabilities to a tool originally designed for configuration management. Ansible allows the use of both declarative and

TABLE II
RESULTS

Number of nodes	Manual provisioning time, min	Ansible provisioning time, min
50 Cirros + 1 Ubuntu18.04	≈ 22	≈ 11
100 Cirros + 1 Ubuntu18.04	≈ 32	≈ 21

imperative programming approaches and YAML for configuration is easy to start with. Finally, its server-only architecture decreases the amount of software and configuration that need to be deployed and maintained.

The purpose of our evaluation was to obtain a first impression on how it is to use Ansible in practice and to observe any limitations that the qualitative study from the previous section failed to capture. We have done this by implementing the second use-case described in Section III (*i. e.,* the IPv6 network scanning) on our test infrastructure. The infrastructure is OpenStack-based and consists of one controller node and two compute nodes. All nodes are equipped with Intel(R) Xeon(R) E3-1220 4 Cores and 20 GB RAM.

The use-case is implemented by starting a set of CirrOS Linux VMs and one instance of Ubuntu18.04 VM (Linux Kernel version 4.15). The CirrOS VMs are allocated IPv6 addresses. The IPv6 scanning algorithms are installed on the Ubuntu18.04 VM. Then, the IPv6 scanning is started and the goal is to discover the IPv6 addresses allocated to the CirrOS VMs. The entire flow will be first implemented manually, and then automatically using an Ansible playbook. This experiment is done first with 50 CirrOS VMs and then repeated with 100 CirrOS VMs.

The results of experiment are shown in Table II. They indicate that with the usage of Ansible it took approximately 1.5–2 times less time to deploy the instances, run the IPv6 scanning algorithms and get the output.

In general, we were satisfied with the Ansible approach to implement IaC. However, due to its server-only architecture we have concerns regarding its scalability when the number of clients increases. These concerns will be addressed as part of our feature work. In addition, we are currently implementing the AI pipeline use-case (*i. e.,* the AI pipeline) as a second evaluation for Ansible and our test infrastructure.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the funding projects that are supporting the work presented in this paper: a) The work was partly sponsored by the *Swedish Agency for Economic and Regional Growth (Tillväxtverket)* under the umbrella of European Structural and Investment Funds (ESI) covered by grant agreement No 20201213 (Test Arena Blekinge). The project is run in collaboration with Blue Science Park in Karlskrona, Sweden. b) In addition, this work has received partial funding from the European Unions Horizon 2020 research and innovation program under grant agreement No 732204 (Bonseyes). This work is supported by the Swiss State Secretariat for Education Research and Innovation (SERI) under contract number 16.0159. The opinions expressed and arguments employed herein do not necessarily reflect the official views of these funding bodies.

The authors would also like to thank Mårten Holmberg from Blue Science Park for his support in defining the test infrastructure.

REFERENCES

[1] Amazon Web Services Inc. [Online]. Available: https://aws.amazon.com
[2] K. Morris, *Infrastructure As Code: Managing Servers in the Cloud*, 1st ed. O'Reilly Media, Inc., 2016.
[3] ETSI, "Network functions virtualisation (NFV) release 2; management and orchestration; network service templates specification," in *ETSI GS NFV-IFA 014 V2.5.1 (2018-08)*, Group Specification, Available at www.etsi.org, 2018.
[4] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc *et al.*, "TFX: A tensorflow-based production-scale machine learning platform," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2017, pp. 1387–1395.
[5] M. de Prado, J. Su, R. Dahyot, R. Saeed, L. Keller, and N. Vallez, "AI Pipeline – Bringing AI to You," Submitted; available on request from: https://www.bonseyes.com/, 2018.
[6] D. Crankshaw, G.-E. Sela, C. Zumarn, X. Mo, J. Gonzalez, I. Stoica, and A. Tumanov, "InferLine: ML inference pipeline composition framework," *arXiv preprint arXiv:1812.01776*, 2018.
[7] M. Chiosi, D. Clarke, P. Willis, A. Reid, J. Feger, M. Bugenhagen, W. Khan, M. Fargano, C. Cui, H. Deng *et al.*, "Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges and Call for Action," in *SDN and OpenFlow World Congress*, vol. 48. sn, 2012.
[8] SDX Central, "What is network service chaining? definition," 2018. [Online]. Available: https://www.sdxcentral.com/networking/virtualization/definitions/what-is-network-service-chaining
[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proceedings of USENIX OSDI*, 2016, pp. 265–283.
[10] Bonseyes Project, "Bonseyes – Artifical Intelligence Market Place," 2018. [Online]. Available: https://www.bonseyes.com
[11] V. A. Mehri, D. Ilie, and K. Tutschku, "Privacy and DRM requirements for collaborative development of AI applications," in *Proceedings of ARES*, Hamburg, Germany, Aug. 2018.
[12] ETSI OSM, "Open source MANO," 2019. [Online]. Available: https://osm.etsi.org/
[13] ——, "Open source MANO information model," 2019. [Online]. Available: https://osm.etsi.org/wikipub/index.php/OSM_Information_Model
[14] OpenStack, "Welcome to tacker documentation," 2019. [Online]. Available: https://docs.openstack.org/tacker/latest
[15] ——, "Tacker role for openstack-ansible," 2017. [Online]. Available: https://docs.openstack.org/openstack-ansible-os_tacker/latest
[16] OpenStack, "What is openstack?" 2019. [Online]. Available: https://www.openstack.org/software/
[17] P. Arlos, A. Carlson, D. Ilie, and K. Tutschku, "Introduction to XIFI," in *Third National Workshop in Data Science (SweDS)*, 2015. [Online]. Available: https://denver.bth.se/projects/bigdata.nsf/attachments/SweDS-15-Program_pdf/$file/SweDS-15-Program.pdf
[18] OpenStack, "Federated keystone," 2019. [Online]. Available: https://docs.openstack.org/security-guide/identity/federated-keystone.html
[19] V. A. Mehri, D. Ilie, and K. Tutschku, "Designing a secure IoT system architecture from a virtual premise for a collaborative AI lab," in *Proceedings of DISS*, San Diego, USA, Feb. 2019.
[20] E. Bergenholtz, D. Ilie, A. Moss, and E. Casalicchio, "Finding a needle in a haystack – a comparative study of IPv6 scanning methods," in *Proceedings of IEEE ISNCC*, Istanbul, Turkey, Jun. 2019, accepted for publication.
[21] A. Raza, "Puppet vs. chef vs. ansible vs. saltstack." intigua.com, September 2016. [Online]. Available: https://www.intigua.com/blog/puppet-vs.-chef-vs.-ansible-vs.-saltstack
[22] Y. Brikman, "Why we use terraform and not chef, puppet, ansible, saltstack, or cloudformation." blog.gruntwork.io, September 2016. [Online]. Available: https://blog.gruntwork.io/why-we-use-terraform-and-not-chef-puppet-ansible-saltstack-or-cloudformation-7989dad2865c