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

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
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 the 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.

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>Manual provisioning time, min</th>
<th>Ansible provisioning time, min</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 Cirros + 1 Ubuntu18.04</td>
<td>≈ 22</td>
<td>≈ 11</td>
</tr>
<tr>
<td>100 Cirros + 1 Ubuntu18.04</td>
<td>≈ 32</td>
<td>≈ 21</td>
</tr>
</tbody>
</table>

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