Developing a New DevOps Modelling Language to Support the Creation of Infrastructure as Code*

Michele Chiari[1][0000-0001-7742-9233], Elisabetta Di Nitto[1][0000-0003-3422-5171], Adrián Noguero Mucientes[2], Bin Xiang[1][0000-0003-4065-5557]

[1] DEIB, Politecnico di Milano, Milano, Italy, name.surname@polimi.it
[2] Go4IT Solutions, Parque Tecnológico Bizkaia, Bilbao, Spain

Abstract. The deployment of cloud applications and the correct management of their lifecycle is a colossal task. Infrastructure as Code (IaC) tools make this task easier; however, they require the user to have a deep knowledge of both the IaC language and the characteristics of various cloud services providers. The PIACERE project has developed a DevOps Modelling Language (DOML), aiming at describing cloud applications that are agnostic of the specificities of the different providers and IaC tools used for provisioning, deployment and configuration. DOML provides several modeling perspectives in a multi-layer approach. An application can be described in three layers: application layer, abstract and concrete infrastructure layer. It allows developers to describe how cloud applications are structured in an abstract manner, mapping the different software components to the concrete infrastructure elements, enabling the usage of different concretizations to match one particular deployment. This paper provides an overview of the DOML language: its layers and extension mechanisms, as well as an example to showcase its modeling capabilities.

Keywords: Infrastructure as Code, DevOps Modelling Language, multi-layer approach, abstraction

1 Introduction

IaC (Infrastructure as Code) [1] has introduced the possibility to program beforehand the way software is deployed and configured on some execution environment composed of Virtual Machines (VMs) and/or various kinds of containers. Thanks to this IaC programming effort, it is possible to replicate a deployment multiple times by just running a script, to keep the characteristics of the operational environment under control, to better maintain the applications, and to speed up the time to market for a product.

However, building IaC is not a trivial task. It requires an in-depth knowledge of both the IaC language to be used and the characteristics of the target operational environment. In this context, the PIACERE project [2] aims at allowing DevOps teams to model different infrastructure environments, by means of abstractions, through a

* This project has received funding from the European Union's Horizon 2020 programme under grant agreement No 101000162 (PIACERE).

DevOps Modelling Language (DOML) that hides the specificities and technicalities of the current solutions and increases the productivity of these teams. Models defined in the DOML are then translated, through the Infrastructural Code Generator (ICG), into the target languages needed by the existing IaC tools, to reduce the time needed for creating infrastructural code for complex applications.

DOML models are created through the PIACERE IDE, which supports users in their activities through suggestions and guidance and integrates all other design-time PIACERE tools.

Another issue to consider is that, in the current highly dynamic and evolving context, new computing resources as well as new IaC languages and tools are continuously emerging. This requires the definition of proper extensibility mechanisms for the DOML and the corresponding ICGs to ensure the sustainability and longevity of the PIACERE approach and tool-suite. To this end, the DOML Extension mechanisms (DOML-E) will allow new infrastructural components and IaC tools to be incorporated in the DOML language for software execution, network communication, cloud services, or data storage.

2 Current IaC Approaches

The IaC area includes several different languages and runtime environments that focus on specific aspects of the whole problem of automating deployment and runtime management of complex applications. For instance, prominent languages today are Terraform [3] and TOSCA [4], mostly focusing on provisioning of resources in multiple cloud environments, Ansible [5], Chef [6] and Puppet [7] mostly tackling the problem of configuring VMs and on deploying software layers on top of them, the Dockerfile [8] language for controlling the creation of execution containers that can be used on top of any operating system to decouple a software component from low level details, the Kubernetes [9] configuration language to customize the operational environment features that support monitoring, autoscaling, restart of components.

In summary, there is a large variety of competing approaches requiring the adoption of different programming languages for writing infrastructural code. All these are focusing on a single or a small set of automation steps and of resource types (e.g., VMs). They mostly focus on cloud computing, leaving aside other computational resources such as those at the edge. Thus, there is not really an end-to-end solution covering all aspects and developers are forced to use a combination of different languages and tools.

3 DOML Modelling Principles

To address the aforementioned issues, in PIACERE we are developing the DOML as a high-level modelling approach that is mapped into multiple IaC languages addressing specific aspects of infrastructure resources provisioning, and application deployment and configuration. In the following, we present the principles guiding our approach.

3.1 A single model for multiple IaC code fragments

In the definition of the DOML, we aim at enabling users to create models that can result in IaC code written in different languages and dedicated to executing different operations. E.g., let us assume that we create a DOML model corresponding to the UML diagram shown in Fig. 1. Here we adopt the well-known UML notation to formulate examples intuitively. The DOML syntax, however, is not based on UML to avoid coping with its complexity. The diagram shows a component A that requires the installation of NodeJS for its execution. In turn, NodeJS is running on a Docker container on a VM.

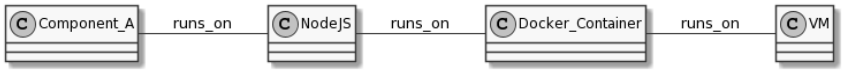

Fig. 1. Relationships between a component and the execution environment it runs on.

We can infer that the following steps must be performed to deploy and run the system:

1. A container image must be created, incorporating NodeJS and component A.
2. A VM with the required characteristics must be provisioned and associated to a public IP address; this step can be executed in parallel with the previous one.
3. A Docker engine must be deployed in the VM.
4. The container image must be run on the VM by the Docker engine.

If the container is properly configured, the web server and component *A* can start their execution.

The above steps can be accomplished if we generate, either manually or automatically, the following artifacts:

- A Dockerfile that manages the creation of the container image (step 1)
- A Terraform or TOSCA blueprint in charge of orchestrating steps 2 to 4, interacting with the VM provider and executing all needed scripts.
- Some Ansible playbooks or similar scripts that execute steps 3 and 4.

Besides the complexity of the individual files to be created, an important issue we note is that these files are all written using different languages featuring a different programming model. With DOML we would like to understand the extent to which the scripts needed to accomplish the above steps can be derived from a high-level model including the components identified in Fig. 1, thus limiting the need for the end users to work with the target languages as much as possible.

3.2 Separation of concerns and multiple modelling layers

Another objective we want to tackle is to support users in separating the modelling of application-level components from the one of their execution environments (e.g., containers, VMs, etc.). The rationale for this choice is that different users, with different skills and roles, could be focusing on these two aspects. Typically, the application designer will focus on the application structure definition in terms of components and their connections (cf. Fig. 2), while an Ops expert will oversee the allocation of components within proper computational elements. The allocation will have to allow the fulfilment of the specified non-functional requirements.

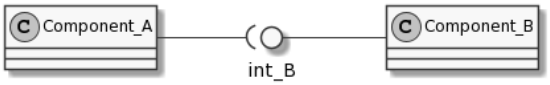

Fig. 2. Modelling the application structure.

Furthermore, given the availability of multiple providers/technologies offering IaaS (Infrastructure-as-a-Service) and, in some cases, compatible PaaS (Platform-as-a-Service) solutions, we want to offer the possibility to provide an abstract definition of the infrastructure (cf. Fig. 3) to be used for an application and, then, to define different concretizations of this same infrastructure, so to support deployment and execution of applications into multiple contexts (see the left- and right-hand side of Fig. 4).

For instance, the same components could be made available in two different deployments: an in-house containerized installation to be used for pre-release testing, and a cloud-based non-containerized installation to be used as the main operational environment, resulting into multiple possible mappings of the same abstract computing node.

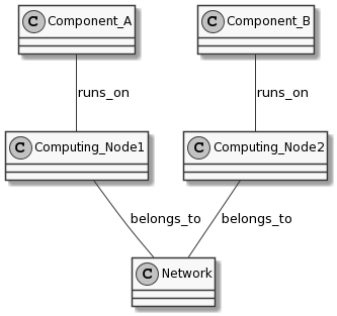

Fig. 3. Modelling an abstract infrastructure and the mapping with components.

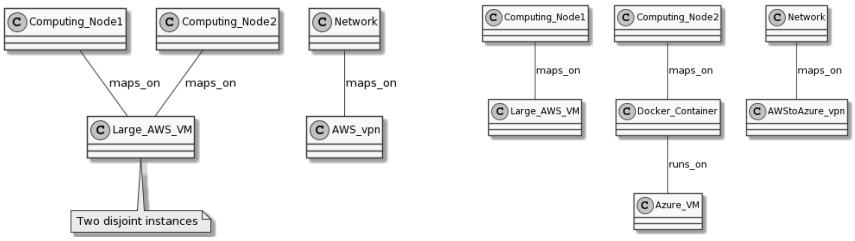

Fig. 4. Modelling different concretizations of an abstract infrastructure.

4 An Example of a DOML Model

Fig. 5 shows the skeleton of a simple example of a DOML model, in which a *nginx* web server runs on a VM provisioned by the OpenStack provider. For space constraints, we do not report the whole code. The model is organized in layers. The *application layer* defines the nginx server instance as a software component. The *infrastructure layer*

defines a VM connected to a network. The *deployment configuration* states that the nginx instance runs on the VM. Finally, the *concretization layer* defines how components from other layers are mapped to services offered by a specific cloud service provider, in this case OpenStack.

```
doml nginx_openstack

application app {
    software_component nginx {
        properties {...}
    }
}

infrastructure infra {
    vm_image v_img {
        generates vm1
    }
    vm vm1 {
        iface i1 {
            address "16.0.0.1"
            belongs_to net1
        }
    }
    net net1 {
        address "16.0.0.0/24"
        protocol "tcp/ip"
    }
}
```

```
deployment config {
    nginx -> vm1
}

concretizations {
    concrete_infrastructure con {
        provider openstack {
            vm concrete_vm {
                properties {...}
                maps vm1
            }
            vm_image con_vm_image {
                properties {...}
                maps v_img
            }
            net concrete_net {
                properties {...}
                maps net1
            }
        }
    }
    active con
}
```

Fig. 5. Example DOML model of a cloud application running a nginx instance.

5 Conclusion

The development of the DOML language is an ongoing effort. The first version has been released together with the IaC code generation tool that is able to create Terraform and Ansible scripts. Currently, we are experimenting with the approach through our case studies.

References

1. Morris, K.: Infrastructure as code: managing servers in the cloud. O'Reilly Media, Inc. (2016).
2. PIACERE Homepage, https://www.piacere-project.eu/, last accessed 2022/03/02.
3. Terraform Homepage, https://www.terraform.io/, last accessed 2022/03/02.
4. TOSCA, OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC. OASIS OPEN, OASIS, last accessed 2022/03/02.
5. Ansible Homepage, https://www.ansible.com/, last accessed 2022/03/02.
6. Chef Homepage, https://www.chef.io/, last accessed 2022/03/02.
7. Puppet Homepage, https://puppet.com/, last accessed 2022/03/02.
8. Dockerfile, https://docs.docker.com/engine/reference/builder/, last accessed 2022/03/02.
9. Kubernetes Homepage, https://kubernetes.io/, last accessed 2022/03/02.