DOML: A New Modelling Approach to Infrastructure-as-Code

Michele Chiari[1], Bin Xiang[2(✉)], Galia Novakova Nedeltcheva[2], Elisabetta Di Nitto[2], Lorenzo Blasi[3], Debora Benedetto[3], and Laurentiu Niculut[3]

[1] TU Wien, Treitlestraße 3, Vienna, Austria
michele.chiari@tuwien.ac.at
[2] Politecnico di Milano, Piazza Leonardo Da Vinci 32, Milano, Italy
{bin.xiang,galia.nedeltcheva,elisabetta.nitto}@polimi.it
[3] Hewlett Packard Italiana s.r.l., Via Giuseppe Vittorio, Cernusco sul Naviglio, Italy
{lorenzo.blasi,debora.benedetto,laurentiu.niculut}@hpe.com

Abstract. One of the main DevOps practices is the automation of resource provisioning and deployment of complex software. This automation is enabled by the explicit definition of *Infrastructure-as-Code* (IaC), i.e., a set of scripts, often written in different modelling languages, which defines the infrastructure and applications to be deployed.

We introduce the DevOps Modelling Language (DOML), a new Cloud modelling language for infrastructure deployments. DOML is a modelling approach that can be mapped into multiple IaC languages, addressing infrastructure provisioning, application deployment and configuration at once. The idea behind DOML is to use a single modelling paradigm which can help to reduce the need of deep technical expertise in using different specialised IaC languages.

We present the DOML's principles and discuss the related work on IaC languages. We demonstrate the DOML advantages for the end-user in comparison with state-of-the-art IaC languages such as Ansible, Terraform, and Cloudify, and show its effectiveness through an example.

Keywords: Infrastructure-as-Code · DevOps · IaC Modelling languages · Multi-layer approach · Evaluation

1 Introduction

Employing Infrastructure-as-Code (IaC) means creating and managing an IT infrastructure, typically composed of computational resources and multiple software layers, by defining and executing code written in some special-purpose programming languages [20].

Defining a whole IT infrastructure deployment through IaC introduces several advantages in terms of repeatability of actions, reusability, and speed. However, it requires deep knowledge of multiple IaC languages and frameworks, since each specific framework is covering a specific aspect of the whole problem [5].

M. Indulska et al. (Eds.): CAiSE 2023, LNCS 13901, pp. 297–313, 2023.
https://doi.org/10.1007/978-3-031-34560-9_18

This causes a steep learning curve for non-technical users and even for expert practitioners migrating from other technologies. Moreover, the selection of a specific set of IaCs, given the peculiarities of each individual language, tends to foster vendor lock-in.

In this paper, we propose a low-code approach to IaC, which makes the creation of infrastructural code more accessible to the designers, developers and operators. We present the DevOps Modelling Language (DOML), which hides the specificity and technicalities of the current IaC solutions.

The DOML allows for a complete specification of a deployment from its applications and software services to the infrastructural components and services supporting them. DOML models are mainly structured in three layers. Specifically, software components (e.g., web servers, databases, etc.) are described in the *application layer*, abstracting away from the infrastructure on which they are supposed to run. Infrastructure components are specified in the *abstract infrastructure layer*, and then linked to the applications they are supposed to host. This layer models infrastructural facilities, such as virtual machines, networks, containers, etc., without referring to their actual concretization in specific technologies (e.g., AWS or OpenStack VMs, Docker containers). This aspect is tackled by the *concrete infrastructure layer*, where the user specifies the infrastructure components offered by the Cloud Service Provider (CSP).

This modelling approach comes out from a careful analysis of related works concerning IaC languages, as well as other Cloud modelling approaches [5] and a critical review of the requirements for the DOML, provided by practitioners from several companies (HP Enterprise, Ericsson and Prodevelop).

Following the idea of generating code from an abstract model that is at the heart of Model Driven Development (MDD) [24], DOML models are turned into actual deployments by the Infrastructural Code Generator (ICG), which produces IaCs executable in the existing and well-supported frameworks. Our first target IaCs are Terraform and Ansible. Nevertheless, the same ICG could be extended to generate other IaC languages to target more applications and CSPs.

In this paper, we present the DOML language and its advantages, and show its effectiveness through case studies. We evaluate our approach by comparing its usage with the direct use of IaC languages such as Terraform and Cloudify. We show that the DOML is complete enough to model a whole deployment by itself, while the other approaches require the simultaneous use of more than one IaC language. Moreover, we show that DOML is generally more concise than the competing approaches.

Paper Structure. In Sect. 2 we review some state-of-the-art IaC approaches, highlighting the motivation behind ours. Section 3 presents a simple case study that will be used as running example. Section 4 outlines the principles behind the DOML, while Sect. 5 defines its modeling abstractions; Sect. 6 presents the IaC generation mechanism. Section 7 compares the DOML with state-of-the-art IaC approaches, and Sect. 8 discusses this evaluation. Finally, Sect. 9 concludes the paper.

2 Related Work

Choosing the right approach for automating the provisioning of computational resources and deployment of application components is not an easy task. In fact, each of the available IaC frameworks covers different parts of the whole problem. As a result, multiple frameworks must be combined, resulting in the need for the DevOps teams to understand all such frameworks. IaC frameworks can be divided into the following four categories.

- *Deployment and configuration management* frameworks focus on automating the installation, setup and life cycle of software applications deployed on top of an existing infrastructure. Examples of such tools are Chef [8], Puppet [22] and Ansible [23]. While they have similar purposes, they are quite different from each other in terms of the defined IaC language and of the corresponding execution semantics.
- *Infrastructure provisioning* frameworks focus on describing the infrastructural topology, defining the virtual or physical infrastructural elements and their configurations, and providing automated means of managing their life cycles. For example, Terraform [16] is a proprietary language with an associated executor. It allows users to define an infrastructure configuration; it keeps track of the actual configuration of the managed infrastructure and, when needed, aligns it with the defined configuration. TOSCA [21], instead, is an OASIS standard modelling language that aims at allowing users to specify any type of IT system through powerful abstraction mechanisms, consisting of abstract *node templates* that can be combined through inheritance. The TOSCA language is adopted by a variety of executors that define its operational semantics in different ways [5,10].
- *Virtualization/Containerization* tools provide automation in building and managing VM or container images. An example of such tools is Docker [11] that has become the de-facto standard for running container-based applications on-premises, in public and private cloud providers [18]. Docker solves issues related to application portability, as containerised applications carry on their dependencies. It lets users define the recipe to build a container image using a custom, domain-specific language.
- *Runtime Orchestration* tools automate the whole life cycle of container-based deployments, including scaling and other management operations. An important representative of this category is Kubernetes [9], which also provides its own IaC language.

In general, managing a complex application, multiple of the mentioned frameworks must be used. For instance, the infrastructure to be provisioned (VMs, network elements, firewalls, etc.) could be modeled and then created with Terraform or TOSCA plus its executors. Ansible (or Chef/Puppet) playbooks could be executed to deploy and configure applications on top of the created infrastructure. Given that most of the application components rely on external preexisting software layers, it is typically advisable to embed all needed elements within some

containers. This calls for the usage of Docker or of a similar approach. Finally, if the user wants to have a dynamic management of the application at runtime, an orchestration framework will have to be adopted.

This scenario clearly requires the experienced users who are proficient with multiple IaC languages and tools, and that are able to take advantage of the ample and scattered offers. An initial approach that aims at reducing the learning curve in adopting any DevOps-relevant platform is presented in [10]. The basic idea is to model DevOps processes, platforms and languages and to exploit these models within the context of low-code environments to let non-experienced users to exploit the defined platforms and languages. Other approaches exploit model-driven engineering in the specific context of IaC development. For example, DICER [4] is focusing on deployment and operation of big data applications. It consists of a UML-based Domain-Specific Language (DSL) and a generator to derive TOSCA code from it. The limit of this approach is that it assumes the existence of additional low level scripts (in Chef or Ansible) taking care of the configuration of applications. Such scripts work underneath and are not exposed to nor modifiable by the DevOps team through the DICER modelling framework. SODALITE [25] is another framework based on TOSCA. Its aim is to offer the support and guidance in the creation of TOSCA blueprints through the usage of its defined DSL. Additionally, it supports the creation of Ansible scripts for deployment and configuration, and exploits semantics reasoning to help the users in the modelling task. Despite this, the SODALITE approach still requires users to be proficient in both Ansible and TOSCA.

A more sophisticated approach is EDMM [30], an Essential Deployment MetaModel. It defines the main concepts that are common to multiple deployment and configuration management frameworks and allows users to exploit such concepts to define application models. Then, through some transformators, EDMM supports the generation of codes in various IaC languages.

In the DOML approach proposed in this paper, we follow the EDMM idea of targeting multiple IaC languages, but we try to extend the scope of the approach beyond deployment and configuration. In particular, at the moment, we are also able to handle infrastructure provisioning, and we plan to support both containerization and runtime orchestration in the next releases of the approach. We offer a single modelling language and a smart IaC generation approach allowing inexperienced DevOps teams to manage all aspects of deployment and operation on different types of infrastructures. So, from the same model we are able to produce IaC code in multiple pre-existing languages.

3 Running Example

To illustrate our approach, we use a simple deployment as a case study. It consists of a website hosted by an instance of the NGINX web server [15] deployed on a VM. A more sophisticated example involving more components will be demonstrated in Sect. 7. This example, though, is representative of typical deployments, because it contains some of the most common components (see Fig. 1 for a component diagram representation). The NGINX server instance is the execution

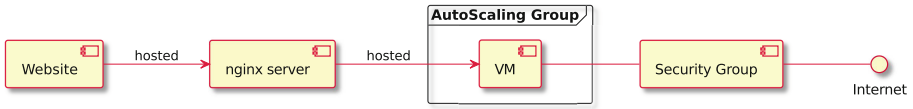

Fig. 1. Component diagram of the NGINX case study.

environment for the website and runs on a VM with a GNU/ Linux-based operating system (Ubuntu 20.04). To ensure the website scalability with respect to the number of connected users, multiple instances of the VM are spawned and managed by an auto-scaling group. The network interface that links the VMs to the Internet is managed by a security group, containing the security rules that enable HTTP, HTTPS and ICMP network traffic. The standard SSH port is enabled, enabling the direct access to the VMs, protected by an RSA key pair for authentication.

An infrastructure like this can be implemented by relying either on a private cloud or on public cloud providers, such as Amazon Web Services, Google Cloud Platform, Microsoft Azure, etc. Initially, in our case study we choose to deploy the application on OpenStack [26], which is an open source industry standard. Further on, in Sect. 7 we show how we can change the underlying provider.

4 DOML Design Principles

In this section, we present the principles underneath the definition of DOML.

4.1 A Single Model for Multiple IaC Fragments

The DOML is defined to support the creation of models resulting in IaC codes written in different languages and dedicated to different operations. For instance, let us consider the system outlined in Fig. 1. The following steps must be performed to deploy the modeled system:

1. A VM with the correct OS must be retrieved if preexisting, or created;
2. The VM must be set up for access through SSH;
3. The NGINX server with the website sources must be installed on the VM;
4. The autoscaling group must be set up with the VM image;
5. The network must be configured with the required security rules;
6. The deployment process must be planned and executed.

To execute the above listed steps adopting the current technologies, we would need some Ansible playbooks or other scripts executing steps 2 and 3, together with a Terraform or TOSCA blueprint to orchestrate all other steps. Such scripts have their inherent complexities, and they are all written in different languages featuring different programming models. With the DOML approach, we aim to derive such scripts from a high-level model, and to reduce the need for the end users to work with the low-level target languages as much as possible.

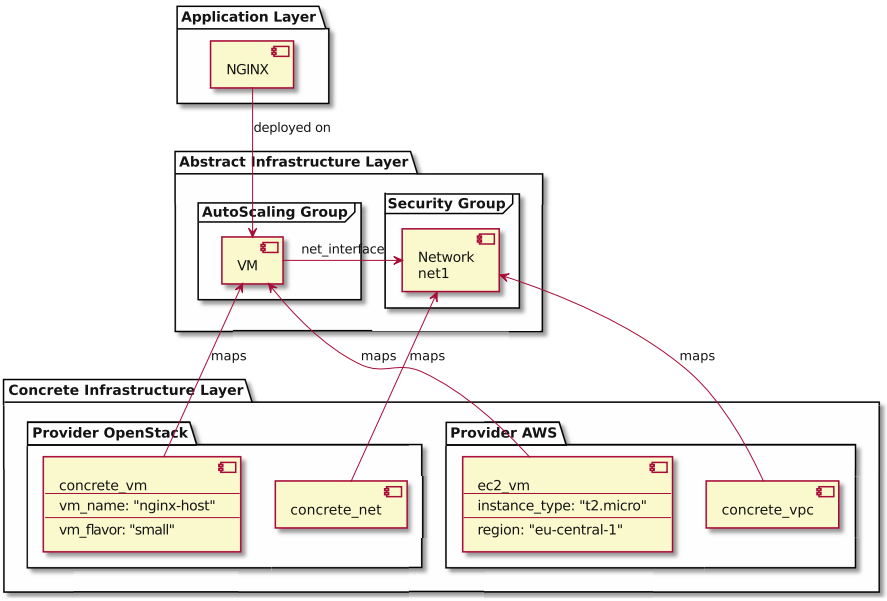

Fig. 2. The NGINX case study represented as different DOML layers.

4.2 Multiple Modeling Layers

Another objective we target is to support separate modeling of the application-level components from their execution environments (e.g., containers, VMs, etc.). In fact, we argue that different users, with different skills and roles, should focus on the specific aspects that fall within their expertise. Typically, the application designer will focus on the application structure definition in terms of software components and their connections, while an operations expert will oversee the allocation of software components within proper computing nodes.

Furthermore, multiple providers and technologies offering the same IaaS (Infrastructure-as-a-Service) and in some cases, compatible PaaS (Platform-as-a-Service) solutions are available. Thus, we want to offer the possibility to provide an abstract definition of the infrastructure to be used to run an application, and then to define different concretisations, so as to support deployment and execution of applications into multiple contexts.

Referring to the example of Fig. 1, Fig. 2 shows a distribution of components into three layers: one describes the application, and the remaining two layers describe the infrastructure at different levels. In particular, the same *abstract* infrastructure can be implemented by two different *concrete* infrastructures, respectively based on OpenStack and AWS.

5 DOML Language

The DOML language implementation consists of two parts: the DOML metamodel, described using Ecore from the Eclipse Modeling Framework (EMF) [13], and the textual syntax used to create models, based on Xtext [14].

We organize all modeled entities in the following layers, which aggregate the modeling abstractions in coherent groups:

- *Application Layer (AL)*: concepts required to define an application, e.g., software components, interfaces, connectors between components, services, and specific subcategories thereof.
- *Abstract Infrastructure Layer (AIL)*: concepts associated to the definition of the infrastructure (e.g., computing nodes) without referencing a specific provider.
- *Concrete Infrastructure Layer (CIL)*: concepts associated to the definition of infrastructure elements within a specific provider, e.g., a Docker container or an Amazon VM.

In Sect. 5.1, we describe the components that can be defined in each layer. We do not include the complete definition of DOML concepts, which is available in [29,31], due to page limits, but we illustrate it in Sect. 5.2 through an example.

5.1 Components of DOML Layers

A *DOMLModel* is composed of an AL, an AIL, one or more CILs, and a *Configuration*. A *Configuration* is a list of one or more *Deployment*s that consist of the associations between *ApplicationComponent*s (from the AL) and *InfrastructureElement*s (from the AIL) on which they are deployed. Only one *Deployment* can be active at a time. All other components derive from a base *DOMLElement* class, which gives them the ability to have custom *Properties* encoding some of their features.

Application Layer (AL). The *ApplicationLayer* is composed of many *ApplicationComponent*s, which can be *SoftwareComponent*s, *SoftwareInterface*s, or *SaaS* (Software-as-a-Service) components. A *SaaS* can be, e.g., a *SaaSDBMS* if it implements a database. *ApplicationComponent*s may expose or consume different *SoftwareInterface*s, providing or requiring services from other *ApplicationComponent*s. For instance, a database *SoftwareComponent* can expose a SQL-based interface, and a web application component can consume it, meaning that the latter will communicate with the former to retrieve and write data.

Abstract Infrastructure Layer (AIL). The *InfrastructureLayer* is composed of *ComputingNode*s, *Network*s, *SceurityGroup*s, and *AutoScalingGroup*s. A *ComputingNode* models any infrastructure element that can run software: it can be a *Container*, a *Physical ComputingNode* or a *VirtualMachine*. *ComputingNode*s can have multiple *NetworkInterface*s that link them to a network. A *Container*

can be generated from a *ContainerImage*, and a *VirtualMachine* from a *VMImage*. A *Network* can have many *Subnet*s, and its configuration is represented by a *SecurityGroup* containing firewall rules.

Concrete Infrastructure Layer (CIL). This layer provides the *concretizations* for the AIL, mapping the abstract infrastructure elements to the concrete ones from the supported cloud service providers. In general, each element of the AIL has a corresponding "concrete" version. The CIL contains one or more *RuntimeProvider*s, e.g., Amazon AWS, OpenStack, etc. Each *RuntimeProvider* contains the concrete elements which are linked to the AIL elements via the *maps* association. For instance, an OpenStack provider could provide *VirtualMachine*s, *Network*s, *Container*s, etc.

5.2 DOML Model of the Running Example

To illustrate the syntax of the DOML, we show and comment the DOML model of the case study of Sect. 3. The entire model can be found in [28].

Application Layer. In Listing 1.1 we show the AL of the DOML model for the deployment of Fig. 1. It only contains a *SoftwareComponent* for the NGINX server, with a *Property* indicating the website's sources.

Listing 1.1. DOML Application Layer

```
application app {
  software_component nginx {
    properties { source_code="/.../html/index.html"; }
  }
}
```

Abstract Infrastructure Layer. The AIL is partially shown in Listing 1.2. It defines the infrastructure topology that supports the execution of application components. We define the VM that hosts the NGINX instance in the autoscaling group that manages it. We declare its guest operating system, its credentials, and its network interface, which is linked to a network called `net1` and controlled by a security group called `sg` (we do not show all components here for space constraints, but they are defined in this layer too [28]).

The *deployment configuration*, at the bottom of Listing 1.2, provides the link between the AL and AIL: it assigns the NGINX instance to the VM.

Concrete Infrastructure Layer. The last step needed to make this DOML model functional is to assign all components in the AIL to a cloud service provider. This is done by defining one or more CILs (only one of which will be active at a time). To simplify the presentation, in Listing 1.3 we show only one of such layers, and only for the VM.

Listing 1.2. Autoscaling group in the AIL, and deployment configuration.

```
infrastructure infra {
  ...
  autoscale_group ag {
    vm vm1 {
      os "ubuntu-20.04.3"
      iface i1 {
        address "10.0.0.1"
        belongs_to net1
        security sg
      }
      credentials ssh_key
    }
  }
}
deployment config {
  nginx -> vm1
}
```

Listing 1.3. Part of the concrete infrastructure layer.

```
concretizations {
  concrete_infrastructure con_infra {
    provider openstack {
      vm concrete_vm {
        properties {
          vm_name = "nginx-host";
          vm_flavor = "small";
          vm_key_name = "user1";
        }
        maps vm1
      }
      ...
    }
  }
  ...
  active con_infra
}
```

We create a concrete infrastructure configuration called `con_infra`. We could assign different components of the AIL to different providers. For the sake of space, in this section we use only OpenStack. Thus, we create a block for the OpenStack provider containing a component for each of the abstract infrastructure elements. In Listing 1.3 we show the VM concretization. Its `maps` attribute links it to the appropriate VM component in the AIL. Moreover, in the CIL we can customize aspects that cannot be described in the AIL because they are provider-specific. Here we choose the name and size of the VM.

The information in this model is enough for the ICG to produce IaC scripts that can create VMs on OpenStack, the autoscaling group and all other required features and installs NGINX on top of such infrastructure.

6 IaC Generation Mechanism

To generate executable IaC code from DOML, we have built a tool named Infrastructural Code Generator (ICG). The ICG receives a DOML model as input and generates IaC code like Terraform, Ansible, etc., as output.

The generation of code from an abstract model is one of the main advantages of MDD [24], but the benefit is real when the generated code is complete and executable, i.e., not just a skeleton or something that needs manual editing. Generating code from a model can be done using Template-Based Code Generation (TBCG), a technique that transforms input data into structured text by using templates [7]. The process is simple: our template engine uses templates that contain code in the target language and substitutes values taken from the input for placeholders. Each template has a static part, that is transferred as-is in the output, and a dynamic part whose result depends on the input. Most template engines support control structures in the dynamic part of their templates, which allows part of the transformation logic to be embedded in the template itself.

Template engines can be classified according to the input they rely on: according to Luhunu [17], there are model-based engines, such as Acceleo [12], that are

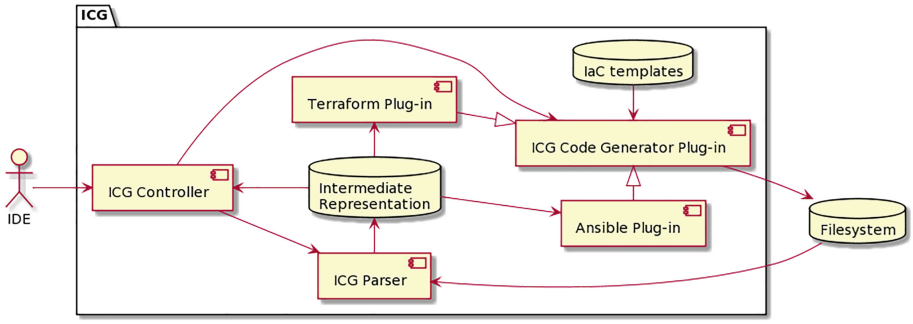

Fig. 3. Infrastructural Code Generator Architecture.

based on an input metamodel, and code-based tools, such as Velocity [3], which rely on a DSL to express the dynamic part of their templates. In our case, the structured text generated is the IaC in languages that can be executed by the most used IaC tools, e.g., Terraform and Ansible. The ICG is based on the DOML metamodel but uses the code-based Jinjia2 library [27]. Jinjia2 is a simple but powerful template engine that supports plain placeholder substitution, and also several control structures, such as loops, conditionals and functions, to build dynamic templates (see Listing 1.4).

We implemented the ICG in Python. Its internal architecture, represented in Fig. 3, is inspired by the classic structure of a compiler (see e.g. [1]) and consists of separate modules for parsing the input and for generating the output, with an Intermediate Representation (IR) in between. The parser reads the DOML model using the PyEcore library [19] and generates an IR as a JSON document. Then, different Code Generator plug-ins, one for each language to be generated, read the input data from the IR and substitute values in the templates. The whole flow is driven by the Controller, that selects the right templates and activates the corresponding plug-in, depending on the information included in the IR itself.

The IR created by the Parser is structured as a sequence of steps, representing the main code blocks to be generated. Each step includes general information, such as the target language, the target cloud provider, the type of DOML object for which code should be generated, and attributes specific to the target DOML object, in the form of key/value pairs, to be substituted in the template for the corresponding placeholder.

The Controller selects the template to be used depending on the information indicated above (target language, cloud provider, and type of DOML object), and then activates the Code Generator plug-in specific to the desired target language. Template selection and plug-in activation are repeated for each one of the steps in the IR. Thus, the ICG starts creating IaC code for the elements in the CIL and navigates up in the model to find other resources for which there is enough information for generating the related code.

When generating code for the example of Fig. 1, the ICG first learns from the CIL of Listing 1.3 that the VM should be deployed on OpenStack. Thus, it

Listing 1.4. ICG template for OpenStack VMs.

```
resource "openstack_compute_instance_v2" "{{␣infra_element_name␣}}" {
  name        = "{{␣name␣}}"
  image_name  = "{{␣os␣}}"
  flavor_name = "{{␣vm_flavor␣}}"
  key_pair    = openstack_compute_keypair_v2.{{ credentials }}.name
  network { ... }
}
```

selects the template shown in Listing 1.4, and populates the fields it finds in the CIL (e.g., `name` with the value of `vm_name`, etc.). The remaining fields (network and key-pair) are populated by looking at the AIL.

The generated IaC code is intended to create an infrastructure in the selected provider or to configure it in some way, in short to modify the target environment. Our aim is to create a stateless code generator, i.e., one that does not take into consideration the current status of the target environment. This works well when the target language is declarative, such as Terraform, but it is not the case when generating code for an imperative language. In this last case, the target code (its template actually) should be idempotent, i.e., such that the status of the target system does not change even if the code is run multiple times.

The current version of the ICG can generate both Terraform and Ansible code, depending on the specific activity that is intended to be performed. We use Terraform for provisioning and Ansible for configuration. The aim is to have a code generator that can be configured to produce code also for other IaC tools, and can be extended also to support new model abstractions: this is partially true for the current version, and will be one of the main targets for the future versions.

7 Evaluation

We evaluate the DOML against other state-of-the-art IaC approaches to demonstrate its capabilities and shortcomings. We provide a discussion of the evaluation in Sect. 8. We identify the following two research questions (RQs):

- RQ1: Can a DOML model represent the information required to generate executable IaC tackling both provisioning and configuration? Is a DOML model more readable and easier to use than the state-of-the-art approaches?
- RQ2: Is a DOML model able to target multiple execution platforms?

We answer these questions by comparing different representations of system larger than the NGINX example and taken from previous literature. The example has been introduced in [6] and is described in the component diagram of Fig. 4.

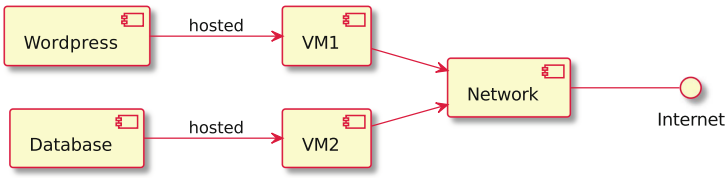

Fig. 4. Component diagram of the Wordpress application deployment from [6].

The application to be deployed is the Wordpress content management system, which runs on a VM. Wordpress depends on a database hosted on a separate VM. The two VMs communicate with each other as well as with the Internet through a common network.

7.1 RQ1: Ability to Generate Executable IaC and Comparison

We answer RQ1 by writing a DOML specification of the Wordpress deployment from Fig. 4 targeting OpenStack as the CSP and successfully running the corresponding IaC code generated by the ICG.

Next, we compare the DOML specification with two equivalent specifications in state-of-the-art languages: the Terraform and Cloudify implementations written by the authors of [6].

DOML. We do not show the whole DOML model for the Wordpress deployment due to the page limits, but it is available in [28]. Similarly to the NGINX example, this DOML model closely resembles the diagram of Fig. 4. The model is small enough to be represented in a single file, and its subdivision into components is very natural: roughly, two components in the application layer for the Wordpress and database, two corresponding VMs and one network in the abstract infrastructure layer, etc.

An advantage brought by the DOML is that the ICG is capable of automatically generating certain common components from its templates, so that they need to be specified in the DOML only if a non-default configuration is required. One of such components is the Security Group: in this Wordpress case study, the ICG creates it automatically thanks to the information included in the AL, so it is not necessary to include it in the DOML model. In fact, the Wordpress component has to be exposed to the Internet by default, and the AL states that it needs to be able to communicate with the database: the security group is created accordingly.

Terraform [6]**.** The Terraform definition of the deployment is centered around the provisioning of VMs. First, the provider is set, in this case AWS. This enables the use of AWS-specific components, such as `aws_instance` to define the VMs. The VMs are defined in this way, with their features set as properties (e.g., size, image to be run, etc.); the network to which VMs are attached is defined similarly. The applications (Wordpress and the database) need to be deployed

Table 1. Metrics on the IaC in Sect. 7.1.

Approach	#LOC	# Files	# Languages	Available at
DOML	103	1	1	[28]
Terraform + Shell	305	3	2	[6]
Cloudify + Ansible	506	9	2	[6]

to their respective VMs. Terraform does not support application deployment directly, but a configuration language—Bash scripts in this case—is needed. The two bash scripts, one for Wordpress and one for the database, are stored as templates, that are instantiated, sent to the VMs through SSH, and automatically executed during deployment.

Cloudify [6]**.** The Cloudify definition of the deployment starts by importing plugins related to the targeted cloud platform. These allow to use provider-specific node types to define the VMs and the networks. Again, the Ansible configuration language is needed to deploy applications to the VMs.

In Table 1 we provide a summary of the objective metrics we have collected.

7.2 RQ2: Targeting Multiple Execution Platforms

DOML supports multi-platform deployment and operation. We show this by modifying the OpenStack-based Wordpress DOML model to incorporate a new deployment on AWS EC2 [2].

The differences in modelling the deployment on AWS EC2 only occur in the CIL and concern the definitions of the VM and the network. In the CIL, the VM image name is specified by the Amazon Machine Images (AMI) format, e.g., `name ="ami-xxx"`, which provides the information required to launch a VM instance. In the VM block, we specify other vendor-specific settings, such as the location information (e.g., `region = "eu-central-1"`) and the `instance_type`. These parameters are optional, and the ICG chooses default values if they are not specified. Another issue caused by switching to AWS concerns the network. Amazon provides Virtual Private Cloud (VPC) for controlling the virtual networking environment, which requires defining a subnet into any network. If no subnet has been defined in the AIL, the ICG creates a default one. This way, a user not familiar with AWS requirements can still create a working deployment.

Listing 1.5 shows the new CIL, while everything else is unchanged. Providers can be switched by just changing the active one.

Listing 1.5. VM definition in the concrete infrastructure layer.

```
concrete_infrastructure
    ↪ con_aws_infra {
  provider aws {
    properties { }
    vm concrete_vm1 {
      properties {
maps vm2
    }
    net concrete_net {
      properties {}
      maps net1
        vm_flavor = "t2.micro";
      }
      maps vm1
    }
    vm concrete_vm2 {
      properties {
        vm_flavor = "t2.micro";
      }

    }
  }
  active con_aws_infra
}
```

8 Discussion

The experiments in Sect. 7 show that DOML and the other languages have a few important differences in their modelling approach.

Firstly, in Terraform and Cloudify everything depends on the chosen cloud provider. Each provider brings in a custom set of resources that define different pieces of the infrastructure, but there is no way of abstracting over them. Thus, when changing the provider, the user must learn the components offered by the target provider and its features, and rewrite most of the deployment configuration. Conversely, in DOML the infrastructure description is completely decoupled from providers, and users need not be familiar with what a specific provider offers. The ICG gathers the information needed to deploy an infrastructure configuration on a specific provider, still allowing for customization in the CIL. This also improves reusability of infrastructure components.

Moreover, Terraform and Cloudify are centered around infrastructure components, while in DOML applications have a central role. In languages other than DOML, the fact that Wordpress is deployed on the VM can only be inferred by reading an Ansible or shell script. In DOML, Wordpress is a first-class component, and it is up to the ICG to ensure that it is correctly deployed on a VM, instead of the user. This makes the whole deployment more robust.

Table 1 shows the size of the considered deployment specifications. The one in DOML is significantly more compact in terms of number of lines of code and files. All other approaches require the use of two languages, one for infrastructure definition and one for configuration management. The DOML, on the contrary, can handle both.

Finally, in Sect. 7.2 we show that the abstraction mechanisms offered by the DOML allow for changing the target cloud platform by just adding and activating a new concrete infrastructure. In Terraform and Cloudify all components are defined in a different way based on the target cloud provider, so changing it requires rewriting most of the specification.

Limitations and Threats to Validity. The experimental evaluation we conducted still has some limitations that will be addressed through further research.

In particular, we compared the DOML definition of the Wordpress deployment with some specific definitions. The main conclusion we inferred from the evaluation is that such definitions are more concise than the DOML-based one. This represents a threat to internal validity, because the analysis of a few specifications does not allow us to exclude the possibility of making more concise ones. Nonetheless, such specifications were written by experts, and we argue that it is unlikely that such a large conciseness gap can be recovered.

Another threat to internal validity is that we only compare metrics concerning code size as a proxy of both ease of writing and maintaining code. However, a proper evaluation of such features would require an empirical study.

The main threat to external validity is that the evaluation is performed on a single application deployment, so the claims we make could not be generalized to the other more complex cloud applications. One of our next research steps will be the evaluation on a larger benchmark suite.

9 Conclusion

We have presented the DOML, a novel approach to cloud deployment modelling. We have shown that the approach works for relatively simple but complete systems with practical significance.

Our next challenge is to check whether the approach is usable and works with more complex case studies, including applications with multiple components that rely on different computational resources and middleware layers.

Our ultimate aim is to be able to write the DOML model only once and then use it to deploy the same complex system on different cloud service providers or physical machines. This has resulted in the definition of the DOML as a multi-layer modelling language, where the application and abstract infrastructure layers include a platform-independent specification of the application and its underlying infrastructure, while the concrete infrastructure layer specifies the details associated to the actual deployment on a specific platform.

The DOML has been developed keeping in mind the need for extensibility, and includes an extension mechanism called DOML-E, which will be analysed in detail in a future work.

Acknowledgments. This work has been funded by the EU Commission in the Horizon 2020 research and innovation programme under grant agreement No. 101000162 (PIACERE project), and partially funded by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT19018].

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, & Tools. Pearson Education India, Noida (2007)
2. Amazon Web Services Inc: AWS EC2 (2022). https://aws.amazon.com/ec2/
3. Apache software foundation: the apache velocity project (2022). https://velocity.apache.org/

4. Artac, M., Borovšak, T., Di Nitto, E., Guerriero, M., Perez-Palacin, D., Tamburri, D.A.: Infrastructure-as-Code for data-intensive architectures: a model-driven development approach. In: Proceedings ICSA 2018, pp. 156–165 (2018)
5. Bergmayr, A., et al.: A systematic review of cloud modeling languages. ACM Comput. Surv. **51**(1) (2018). https://doi.org/10.1145/3150227
6. Rebouças de Carvalho, L., Favacho de Araújo, A.P.: Performance comparison of terraform and cloudify as multicloud orchestrators. In: Proceedings CCGRID 2020, pp. 380–389. IEEE (2020). https://doi.org/10.1109/CCGrid49817.2020.00-55
7. Chared, Z., Tyszberowicz, S.S.: Projective template-based code generation. In: CAiSE Forum, pp. 81–87 (2013) https://doi.org/10.1109/ACIT-CSII-BCD.2016.023
8. Chef: Chef infra: powerful policy-based configuration management system software (2022). https://www.chef.io/products/chef-infra
9. CNCF: Kubernetes (2022). https://kubernetes.io/
10. Colantoni, A., Berardinelli, L., Wimmer, M.: DevOpsML: towards modeling DevOps processes and platforms. In: Proceedings. MODELS 2020, ACM (2020)
11. Docker: docker (2022). https://www.docker.com/
12. Eclipse foundation: acceleo home page (2022). https://www.eclipse.org/acceleo/
13. Eclipse foundation: eclipse modeling framework (EMF) (2022). https://www.eclipse.org/modeling/emf/
14. Eclipse foundation: Xtext (2022). https://www.eclipse.org/Xtext/
15. F5 Inc: nginx (2022). https://www.nginx.com/
16. HashiCorp Inc: Terraform documentation (2022). https://www.terraform.io/docs
17. Luhunu, L., Syriani, E.: Comparison of the expressiveness and performance of template-based code generation tools. In: Proceedings SLE 2017, pp. 206–216. ACM (2017)
18. Miell, I., Sayers, A.: Docker in Practice. Simon and Schuster, New York (2019)
19. Pagel, M., et al.: PyEcore/PyEcore: a Python(NIC) implementation of EMF/Ecore (Eclipse Modeling Framework) (2022). https://github.com/pyecore/pyecore
20. Morris, K.: Infrastructure as Code. O'Reilly Media, Sebastopol (2016)
21. OASIS standard: topology and orchestration specification for cloud applications version 1.0 (2013)
22. Puppet Inc: puppet (2022). https://puppet.com/
23. Red Hat Inc: ansible documentation (2022). https://docs.ansible.com/
24. Selic, B.: The pragmatics of model-driven development. IEEE Softw. **20**(5), 19–25 (2003)
25. SODALITE Consortium: SODALITE: software defined application infrastructures management and engineering (2022). https://sodalite.eu/
26. The OpenStack project: OpenStack (2022). https://www.openstack.org/
27. The pallets project: Jinja (2022). https://jinja.palletsprojects.com/
28. The PIACERE Project: IaC artefacts repository (2022). https://github.com/michiari/DOML-case-study
29. The PIACERE project: PIACERE DevSecOps modelling language (DOML) (2022). https://www.piacere-doml.deib.polimi.it/
30. Wurster, M., et al.: The EDMM modeling and transformation system. In: Yangui, S., et al. (eds.) ICSOC 2019. LNCS, vol. 12019, pp. 294–298. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45989-5_26
31. Xiang, B., Di Nitto, E., Nedeltcheva, G.N.: Deliverable D3.2 PIACERE abstractions, DOML and DOML-E - v2 (2023). https://doi.org/10.5281/zenodo.7645687