Modeling Cloud Infrastructure Provisioning: A Software-as-a-Service Approach

Sandobalin, Julio [1,*] ; **Iñiguez-Jarrín, Carlos** [1]

[1] *Escuela Politécnica Nacional, Departamento de Informática y Ciencias de la Computación, Quito, Ecuador*

Abstract: Provisioning means making an infrastructure element, such as a server or network device, ready for use. DevOps community leverages the Infrastructure as Code (IaC) approach to supply tools for cloud infrastructure provision. However, each provisioning tool has its scripting language, and managing different tools for several cloud providers is time-consuming and error-prone. In previous work, we presented a model-driven infrastructure provisioning tool called ARGON, which leverages the IaC approach using Model-Driven Engineering. ARGON provides a modeling language to specify cloud infrastructure resources and generates scripts to support cloud infrastructure provisioning orchestration. Since ARGON runs in the Eclipse Desktop IDE, we propose to migrate from an ARGON Desktop to an ARGON Cloud as a Software-as-a-Service approach. On the one hand, we developed a domain-specific modeling language using JavaScript Frameworks. On the other hand, we used a Model-to-Text transformation engine through a REST web service to generate scripts. Finally, we carried out an example by modeling infrastructure resources for Amazon Web Services and then generating a script for the Ansible tool.

Keywords: Infrastructure as Code, Domain-Specific Language, Cloud Infrastructure Provisioning, Software-as-a-Services, Model-Driven Engineering

Modelado del Aprovisionamiento de Infraestructura en la Nube: Un Enfoque de Software como un Servicio

Resumen: Aprovisionar significa hacer que un elemento de la infraestructura, como un servidor o un dispositivo de red, esté listo para su uso. La comunidad DevOps aprovecha el enfoque de Infraestructura como Código (Infrastructure as Code, IaC) para proporcionar herramientas para el aprovisionamiento de infraestructura en la nube. Sin embargo, cada herramienta de aprovisionamiento tiene su propio lenguaje de secuencias de comandos, y administrar diferentes herramientas para varios proveedores de la nube lleva mucho tiempo y es propenso a errores. En trabajos anteriores, presentamos una herramienta de aprovisionamiento de infraestructura dirigida por modelos llamada ARGON, que aprovecha el enfoque de IaC mediante la Ingeniería de Software Dirigida por Modelos. ARGON proporciona un lenguaje de modelado para especificar los recursos de la infraestructura de la nube y genera scripts para apoyar la orquestación del aprovisionamiento de la infraestructura de la nube. Dado que ARGON se ejecuta en el IDE de escritorio Eclipse, proponemos migrar de un ARGON Desktop a un ARGON Cloud como un enfoque de Software-como-un-Servicio. Por un lado, desarrollamos un lenguaje de modelado específico de dominio utilizando marcos de trabajo de JavaScript. Por otro lado, utilizamos un motor de transformación de modelo-a-texto a través de un servicio web REST para generar scripts. Finalmente, llevamos a cabo un ejemplo modelando recursos de infraestructura para Amazon Web Services y luego generamos un script para la herramienta Ansible.

Palabras claves: Infraestructura como Código, Lenguaje Específico de Dominio, Aprovisionamiento de Infraestructura en la Nube, Software como un Servicio, Ingeniería de Software Dirigida por Modelos

1. INTRODUCTION

DevOps (*Development and Operations*) promotes collaboration between developers and operations staff using a set of values, principles, practices, and tools to optimize software delivery time (Farley et al., 2010). The continuous deployment (CD) practice generates a lot of attention when a critical defect comes to the production environment or software artifacts are delivered late. Furthermore, the CD practice is the borderline between developers and operation staff in the software delivery cycle. In this scenario, practitioners and researchers use the Infrastructure as Code (IaC) to support the CD practice and improve the software delivery time. IaC is an approach to infrastructure automation based on software development practices (Morris, 2016). The idea behind IaC is to

*julio.sandobalin@epn.edu.ec
Recibido: 10/08/2022
Aceptado: 24/08/2023
Publicado en línea: 14/11/2023
10.33333/rp.vol52n2.09

write and execute code to define, update, provision, and destroy the infrastructure (Brikman, 2019).

Cloud Computing comprises hardware-based services, in which hardware management is highly abstracted, and the infrastructure capacity is highly elastic (Buyya et al., 2011). The DevOps community supplies IaC tools to orchestrate cloud infrastructure provisioning. Each IaC tool uses its files or scripts to define the creation, update, execution, and destruction of the cloud infrastructure resources. However, managing scripting languages of different IaC tools for several cloud providers is time-consuming and error-prone. In previous work, we presented ARGON (Sandobalin et al., 2017a), an infrastructure modeling tool for cloud provisioning to address these problems. ARGON leverages the IaC approach through Model-Driven Engineering (MDE). On the one hand, ARGON abstracts the complexity of managing the particularities of cloud providers to define their infrastructure resources using a domain-specific modeling language called ArgonML (*ARGON Modeling Language*). On the other hand, ARGON generates scripts by a Model-to-Text (M2T) transformation engine for different IaC tools to support the orchestration of cloud infrastructure provisioning. ARGON (Sandobalin et al., 2017b) has proven work suitable in a toolchain for cloud infrastructure provisioning using DevOps community tools. Furthermore, ARGON (Sandobalin et al., 2018) supports multi-cloud infrastructure provisioning and proposes a flexible migration process among cloud providers. Finally, a family of controlled experiments (Sandobalin et al., 2020) was carried out to compare ARGON with Ansible as regards their effectiveness, efficiency, perceived ease of use, perceived usefulness, and intention to use.

ARGON works in Eclipse Modeling Framework (Steinberg et al., 2009). Eclipse is an Integrated Development Environment (IDE) with multiple programming languages and modeling tools. Eclipse provides a Desktop IDE and Cloud IDE. The former is installed, launched, and run locally on a computer. The latter is a web-based integrated development platform that can be accessed from different web browsers. Eclipse Desktop IDE has been used for a long-term period. Nevertheless, Eclipse Desktop IDE has all problems of desktop applications, such as portability, being limited by hardware, being restricted to just one machine, requiring the user to download and install it before using, etc. In contrast, Eclipse Cloud IDE works through a web browser with advantages such as users can access through all their devices, portability, less hardware dependence, fewer hardware and software compatibility issues, updates and upgrades being more accessible, etc.

Since the advent of Cloud Computing, many software applications have migrated to one of its services. In this scenario, Eclipse Cloud IDE provides its environment as a Software-as-a-Services (SaaS). However, the Eclipse Cloud IDE does not provide all the programming languages and modeling tools that offer Eclipse Desktop IDE. Therefore, our main research objective is to migrate ARGON toward Cloud Computing to provide infrastructure modeling and script generation as a SaaS service. In this scenario, we migrate ARGON Desktop IDE toward ARGON Cloud IDE to provide a SaaS approach. We did not use Eclipse Cloud IDE because it does not provide the necessary tools for developing domain-specific modeling languages and M2T transformation.

The remainder of this paper is structured as follows: Section 2 discusses related works and identifies the need for modeling the infrastructure resources as a SaaS service. Section 3 presents ARGON Cloud as a SaaS approach. We explain the development of a domain-specific modeling language using JavaScript Frameworks and an M2T transformation engine to generate scripts employing a REST web service. Section 4 introduces an illustrative case study that shows the feasibility of the ARGON Cloud for modeling infrastructure resources and generating scripts. Finally, Section 5 presents our conclusions and future work.

2. RELATED WORK

There has been considerable interest in managing cloud infrastructure provisioning in recent years, and several approaches and strategies have emerged to support it. In this scenario, cloud providers have tools to define, update, manage, execute, and destroy their infrastructure resources, for instance, CloudFormation and OpsWorks of Amazon Web Services. Moreover, the DevOps community has developed several tools to manage the infrastructure provisioning of different cloud providers, such as Terraform and Ansible, and tools to install and manage software in existing servers, such as Chef and Puppet.

Solayman et al. (2023) propose an approach to automate the provision and orchestration of IoT components on Edge and Cloud Computing. In this scenario, the authors use DevOps tools to provision and orchestrate container-based IoT applications through practices such as continuous integration and deployment.

Neharika et al. (2023) investigate secure infrastructure provisioning. This approach achieves secure and automatic infrastructure provisioning using a source code analysis tool, container security tool, and Infrastructure as Code (IaC) tools. The IaC scripts of containers are scanned, and when critical vulnerabilities are not found, the infrastructure is automatically provisioned using Terraform tool.

Miñón et al. (2022) present the Pangea tool to generate suitable execution environments for deploying analytic pipelines. The pipelines are executed on edge, fog, cloud, or on-premise computing settings. First, Pangea provisions infrastructure on demand if it does not exist. Subsequently, Pangea configures each host operative system, installs dependencies, and downloads the code to execute. Finally, Pangea deploys the pipelines.

Kartheeyayini et al. (2022) propose an approach for DevOps tools management using a Domain-Specific Language (DSL) based on IaC. The DSL model the final state of a provisioning infrastructure on the cloud and generating the provisioning scripts for Amazon Web Services (AWS). The authors propose moving legacy applications to a cloud platform by creating infrastructure as code dynamic infrastructure platforms to deploy and manage different applications design such as microservices applications, IoT applications, legacy applications, etc.

Chiari et al. (2022) present a DevOps Modelling Language (DOML) to describe cloud applications of different cloud providers and IaC tools. DOML supplies different modeling perspectives in a multi-layer approach. DOML describes an application, abstract, and concrete infrastructure layer. This approach supports developers in abstractly defining cloud applications, mapping software components to infrastructure elements, and deploying the software application.

Palma et al. (2022) present DEFUSE, a language-agnostic tool

for software defect prediction. DEFUSE tool collects and classifies failure data to enable data correction and then builds machine learning models to detect defects based on the data classified. DEFUSE tool supports the IaC practice to enable infrastructure provisioning through the definition of machine-readable files.
López-Viana et al. (2022) present a proof of concept based on Cloud Native Computing Foundation (CNCF) tools to check the feasibility of using GitOps with the Internet of Things (IoT) along with Edge computing (EC) solutions. The authors found several drawbacks to using these tools, such as a lack of automatic infrastructure provisioning and limitations on the edge devices that can be supported. Additionally, the authors aim to replicate best practices used in cloud-native development and operation.
Zhou et al. (2019) designed and implemented a framework called CloudsStorm to allow developers to easily leverage different cloud providers' virtual infrastructure functions (VIF) for programming their cloud applications. CloudsStorm supports application and infrastructure programmability at the design, infrastructure, and application levels. Besides, Clouds Storm also defines infrastructure provisioning using IaC tools.
Guerriero et al. (2019) present the state of adopting the IaC practice and the critical software engineering challenges. The authors carried out semi-structured interviews with senior developers. The study shows how practitioners adopt and develop the IaC practice for infrastructure provisioning. Moreover, the study also presents the advantages and disadvantages of using IaC tools and the practitioner's needs when dealing with developing, maintaining, and evolving infrastructure provisions using IaC practice. The findings establish that the currently available IaC tools are still limited, and developers feel the need for novel techniques for maintaining IaC code.
Ferry et al. (2018) present a Cloud Modeling Framework (CloudMF), which has a domain-specific language for specifying the deployment and provisioning of multi-cloud applications. CloudFM proposes a Cloud Provider-Independent Model (CPIM) to define the provisioning and deployment of cloud applications. Moreover, CloudFM has a Cloud Provider-Specific Model (CPSM) that uses a model@run-time engine to request cloud providers a list of available resources to refine the CPIM into a CPSM.
Casola et al. (2017) provide a DevOps approach to developing multi-cloud applications (MUSA) with Service Level Agreements (SLAs). MUSA has a modeler tool to define application architectures and deployment requirements using a modeling language based on CAMEL (Rossini, 2015) to define application architectures and deployment requirements.
Nitto et al. (2017) propose a model-driven approach for designing and executing applications on multiple Clouds (MODAClouds). On the one hand, Quality of Service (QoS) requirements are defined and specified on the Cloud Independent Model (CIM) level. On the other hand, cloud-specific characteristics are defined on the CPIM level. Consequently, the CPSM level specifies a precise provider and service for the software application, runs QoS analyses and generates appropriate deployment, monitoring, and self-adaptation scripts to support the runtime phases.
Chen et al. (2016) developed a model-driven operation service (MORE) for cloud-based IT systems that automates deployment and dynamic configuration of software applications. MORE supplies an online modeling editor to specify a topology model, deployment structure, and desired state. MORE transforms the topology model into executable code for Puppet to deploy virtual machines, physical machines, and containers.
In summary, current research works have focused on reusing DevOps community tools (i.e., IaC Tools) to breach gaps described in infrastructure provisioning and applications deployment in cloud computing. Furthermore, researchers focus on providing support for modeling and deploying cloud applications and managing the CPIM and CPSM levels following a model-driven approach. In contrast, ARGON Cloud abstracts the issues related to infrastructure modeling of different cloud providers and then generates scripts for DevOps provisioning tools. As a result, ARGON Cloud offers a SaaS approach to provide a Domain-Specific Modeling Language (DSL) and an M2T transformation engine.

3. ARGON CLOUD

ARGON Desktop (Sandobalin et al., 2017a) is an infrastructure modeling tool for cloud provisioning, which uses the fundamental principles of Model-Driven Engineering (MDE): *abstraction* and *automation*. On the one hand, ARGON Desktop provides ArgonML (ARGON Modeling Language) to abstract cloud capabilities and thus model cloud infrastructure resources. On the other hand, ARGON Desktop uses a Model-to-Text (M2T) transformation engine to generate scripts for cloud infrastructure provisioning automatically. The following subsections explain the ARGON Cloud as a Software-as-a-Services (SaaS) approach.

3.1 Domain-Specific Modeling Language

DevOps community provides a vast range of IaC tools for supporting the orchestration of cloud infrastructure provisioning, such as Ansible, Terraform, CloudFormation, OpenStack Head, etc. Each IaC tool has a different scripting language to define the cloud infrastructure resources. We propose ArgonML to mitigate the complexity of different scripting languages and facilitate a holistic infrastructure modeling language. ArgonML is a Domain-Specific Modeling Language (DSL) to model infrastructure resources for cloud providers. ArgonML follows the design principles for developing DSL proposed by Brambilla et al. (2017).

3.1.1 Abstract Syntax

ArgonML has an Infrastructure Metamodel (Sandobalin et al., 2017a) (IMM) to define modeling concepts, relationships, and properties of cloud infrastructure resources. According to Brambilla et al. (2017), a metamodel constitutes the definition of a modeling language since it describes the whole class of models that language can represent. Figure 1 shows an excerpt from the IMM, which abstracts cloud capacities, such as computing, storage, networking, and elasticity of different cloud providers, to obtain a holistic infrastructure modeling language.
ARGON Desktop runs on Eclipse Modeling Framework (Steinberg et al., 2009) (EMF) to provide a modeling environment, and thus, the metamodeling language Ecore* defines the IMM. However, Ecore is restricted to defining metamodels based solely on the EMF environment. In this scenario, to provide an abstract

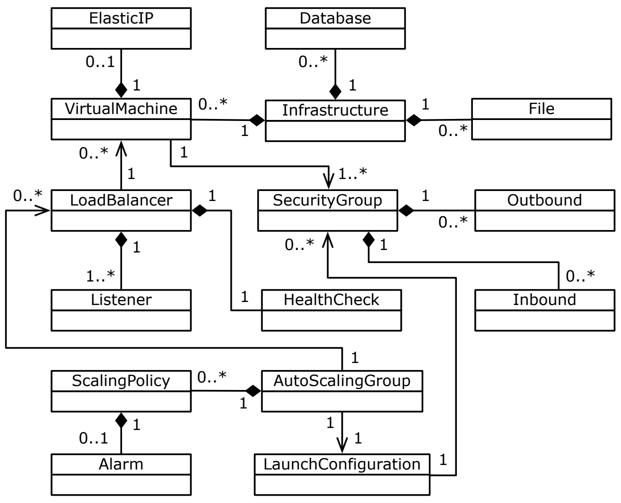


Figure 1. Excerpt from the Infrastructure Metamodel (Sandobalin et al., 2017a)

syntax (i.e., infrastructure metamodel) for ARGON Cloud, we conduct a comparative study of JavaScript Frameworks to obtain the most suitable option to define the IMM in a textual manner. As a result, we chose Ecore.js to represent the IMM metamodeling using JavaScript. Ecore.js implements the Ecore model in JavaScript, including JSON and XMI (XML Metadata Interchange) writers and parsers for web browsers and Node.js.

We describe the main steps below to outline the metamodeling process and thus obtain an infrastructure metamodel.

- **Modeling domain analysis.** Since the specific domain of the DSL is the Infrastructure-as-a-Service (IaaS), we select the principal cloud providers, such as Google Computing Engine, Microsoft Azure, and Amazon Web Services, to define a holistic IMM. We chose these cloud providers due to their education licenses.
- **Modeling language design.** We focus on the infrastructure resources of cloud providers to abstract their capacities (i.e., computing, storage, networking, and elasticity) instead of the particular scripting languages of the DevOps tools. ARGON Cloud also generates the IaC scripts for different DevOps tools from an infrastructure model. As a result, the IMM (see Figure 2a) represents in a textual manner the cloud capacities.
- **Modeling language validation.** To validate infrastructure concepts, the IMM is instantiated for different cloud providers. In this context, we generate an infrastructure model (see Figure 2b) that conforms to the IMM. Therefore, we create dynamic instances to validate concepts, relationships, and properties of infrastructure models, which are in accordance with its IMM.

Figure 2a shows an excerpt from the IMM written in a textual manner using Ecore.js. In Ecore, a Package groups metaclasses and their data types. In this case, the Ecore Package (line 1) defines its name, which is not to be unique. Instead, a URI (Uniform Resource Identifier) uniquely identifies the package. This URI is specified as the value of the nsURI attribute, and the nsPrefix attribute is used to define the corresponding namespace prefix. The Virtual Machine metaclass (line 8) defines its name (line 9) and the supertype (line 10) corresponding to the Element metaclass from which it is extended. The image attribute (line 12) represents a virtual machine image that contains a virtual disk with a bootable operating system. Additionally, we define references (line 24) to other metaclasses, such as the reference to a Security Group metaclass that works as a firewall to the Virtual Machine.

Since Ecore metamodels are serialized using XMI (XML Metadata Interchange). We use the Ecore.js framework to serialize the IMM. Serialization translates a data structure (i.e., IMM) into a format that can be stored, transmitted, and reconstructed in different environments. Figure 2b shows an excerpt from the IMM represented in XMI format. XMI is the format used by the Eclipse environment as a canonical representation of metamodels. Additionally, to export the model (which conforms to its metamodel) is necessary to use the XMI format. For instance, to move an infrastructure model (which conforms to the IMM) toward a Model-to-Text (M2T) transformation engine and thus generate the corresponding IaC script.

The IMM describes the whole infrastructure models that ArgonML can represent. In this case, the IMM defines all infrastructure models expressed using ArgonML. Note that the DSL proposes a holistic modeling language to describe the infrastructure of any cloud provider. Moreover, it is possible to specialize the tags of an infrastructure model for a particular cloud provider, such as Microsoft Azure, Amazon Web Services, Google Computing Engine, etc.

3.1.2 Concrete Syntax

While the *abstract syntax* (i.e., IMM) constitutes the definition of the ArgonML since it describes the whole classes of infrastructure models that can be represented, the *concrete syntax* defines the notation of the graphical language. Several frameworks provide specific languages to describe the concrete syntax for ArgonML, as well as generator components that allow the generation of editors to visualize and manipulate infrastructure models, such as EuGENia, Graphiti, Graphical Editing Framework, etc. Since ARGON Desktop runs on the Eclipse environment, we used EuGENia to generate the corresponding editor to visualize and manipulate infrastructure models. However, ARGON Cloud must be accessed from different web browsers to provide a SaaS approach. Therefore, we also conduct a comparative study of JavaScript Frameworks to obtain the most suitable option to define the graphical notation. As a result, we selected mxGraph, a JavaScript diagramming library that enables create interactive graph applications to run natively in any web browser. According to Brambilla et al. (2017) and considering mxGraph, we define the graphical concrete syntax for ARGON Cloud through particular elements, such as graphic symbols, compositional rules, and mapping.

Figure 3a shows an excerpt from visual editor code using mxGraph. We specify the graphical concrete syntax as follows:

- **Graphical symbols:** The *mxcell class* supports the definition of the virtual machine element (line 1), a visual element to be created in the infrastructure visual editor. In this case, the

a

```
var Package = Ecore.EPackage.create({
    name: 'Infrastructure',
    nsPrefix: 'Infrastructure',
    nsURI: 'platform:/metamodel/Infrastructure.ecore'});
    ...
var VirtualMachine = Ecore.EClass.create({
    name: 'VirtualMachine',
    eSuperTypes: Element
});
var Image = Ecore.EAttribute.create({
    name: 'image',
    eType: Ecore.EString
});
var InstanceType = Ecore.EAttribute.create({
    name: 'instanceType',
    eType: Ecore.EString
});
var Count = Ecore.EAttribute.create({
    name: 'count',
    eType: Ecore.EInt
});
var Groups = Ecore.EReference.create({
    name: 'groups',
    lowerBound: 1,
    upperBound: -1,
    eType: SecurityGroup
});
```

b

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
  name="Infrastructure"
  nsURI="platform:/metamodel/Infrastructure.ecore"
  nsPrefix="Infrastructure">
  ...
  <eClassifiers xsi:type="ecore:EClass"
  name="VirtualMachine"
  eSuperTypes="#//Element">
    <eStructuralFeatures xsi:type="ecore:EReference"
         name="group" lowerBound="1" upperBound="-1"
         eType="#//SecurityGroup">
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
         name="name" eType="ecore:EDataType http://
         www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
         name="image" eType="ecore:EDataType http://
         www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
         name="instance_type" eType="ecore:EDataType
         http://www.eclipse.org/emf/2002/Ecore#//EString"
         />
  </eClassifiers>
  ...
```

Figure 2. Representing the Infrastructure Metamodel textually

a

```
1  var node = doc.createElement('virtualmachine');
2  node.setAttribute('Name', 'Virtual Machine');
3  node.setAttribute('Image', 'ami-0d7c8dde348d3b09f');
4  node.setAttribute('Instance Type', 't2.micro');
5  node.setAttribute('Count', 1);
6  addVertex('images/virtualmachine.png', 40, 40,
          'virtualmachine', virtualmachine);
7
8  graph.multiplicities.push(new mxMultiplicity(true,
          'virtualmachine', null, null, 0, 1,
        ['securitygroup'], null, 'CONNECTION_ERROR' ));
```

b

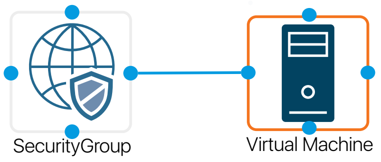


Figure 3. Excerpt from the Concrete Graphical Syntax

virtual machine element has attributes, such as name (line 2), image (line 3), instance type (line 4), and count (line 5).

- **Compositional rules:** The *mxMultiplicity class* helps define how the virtual machine element links (line 8) with the security group element. The link defines how these graphical symbols are nested and combined.

- **Mapping:** Since we are using Ecore.js to define the abstract syntax and mxGraph to create the visual editor (i.e., concrete syntax), we map the graphical symbols to the corresponding abstract syntax elements. The mapping aims to state which graphic symbol should be used for which modeling concept. For instance, a virtual machine element (see Figure 3a) is visualized as a visual element (see Figure 3b).

Note that Figure 3b shows the concrete graphical syntax of a virtual machine. Moreover, virtual machine properties (see Figure 7a) should be defined in this stage. For instance, the Instance Type attribute has t2.micro that will provide 1 vCPU and 1 GB of RAM for a virtual machine in Amazon Web Services.

3.2 ARGON Cloud Architecture

ARGON Cloud follows the metamodeling principle to leverage the IaC approach using MDE. Metamodels can define new modeling languages and properties or features associated with existing information (a.k.a. metadata). Therefore, ARGON Cloud is based on a four-layer MDE architecture. We defined a layered architecture that helps us work at different abstractions levels. The architecture allows us to model the infrastructure resources (independently of any cloud provider and IaC tool) and then generate IaC scripts to orchestrate the cloud infrastructure provisioning. Figure 4 shows the layered ARGON Cloud architecture, where:

- **M3:** The meta-metamodel defines the concepts used at M2 (i.e., metamodel), specifying how to represent the IMM. Since ARGON Cloud will run outside the Eclipse environment, we did not use the Ecore metamodeling language. In this layer, the JavaScript Framework used as a metamodeling language is Ecore.js.

- **M2:** The metamodel defines the concepts used at M1 (i.e., the infrastructure model). In this layer, we specified all infrastructure concepts abstracted from cloud capacities (i.e., computing, storage, networking, and elasticity) using

Ecore.js. The main concepts are infrastructure elements and their attributes. This layer corresponds to the abstract syntax of ArgonML.

- **M1:** At this layer, we model the infrastructure resources based on concepts defined at M2. ARGON Cloud allows modeling infrastructure resources of any cloud provider due to abstraction (at M2) of the capacities of three leading cloud providers (Google Computing Engine, Microsoft Azure, and Amazon Web Services) to define a holistic IMM. As a result, we obtain a generic infrastructure model.
- **M0:** In this layer, an IaC script is the final instance of its infrastructure model. ARGON Cloud automatically generates scripts for several DevOps IaC Tools, such as Ansible, Terraform, etc. ARGON Cloud uses an M2T transformation engine to generate scripts from an infrastructure model. Finally, the IaC tool uses the script to orchestrate the infrastructure provisioning in a particular cloud provider.

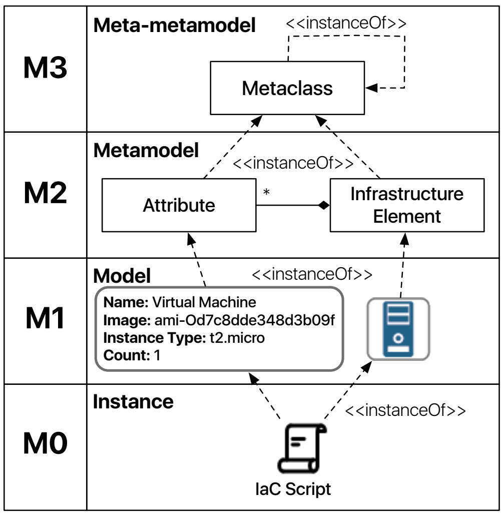


Figure 4. ARGON Cloud Architecture

3.2.1 A Software-as-a-Service Approach

According to Buyya et al. (2011), end-users can access a Software-as-a-Service through web browsers, thus alleviating the burden of software maintenance for customers and simplifying development and testing for providers. ARGON Cloud leverages the SaaS model service to provide end-user access to the visual modeler of cloud infrastructure through a web browser. In this context, ARGON Cloud has two components: a graphical modeler and an M2T transformation engine. The former is ArgonML, a DSL developed using JavaScript Frameworks, such as Ecore.js and mxGraph, to provide the visual modeler of cloud infrastructure. The latter is an M2T transformation engine to support the automatic generation of IaC scripts from infrastructure models.

The M2T transformation engine uses a REST web service to interchange data with the visual modeler. In this case, ArgonML uses the XMI (XML Metadata Interchange format) writer of the Ecore.js library to generate an Ecore XMI file (see Figure 2b) from an infrastructure model. The M2T transformation engine and all its transformation rules were developed using Acceleo language. Sandobalin et al. (2017a) explain the procedure to define the transformation rules.

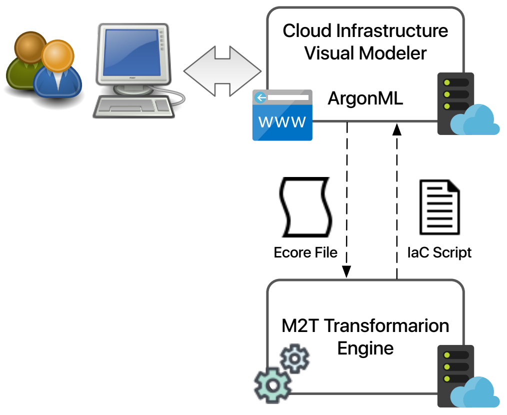

Cloud Infrastructure Visual Modeler
ArgonML
WWW
Ecore File
IaC Script
M2T Transformarion Engine

Figure 5. ARGON Cloud as Software-as-a-Service

On the other hand, Eclipse is a Java-based application requiring Java Runtime Environment or Java Development Kit (JRE or JDK) to run. Therefore, the M2T transformation engine for Eclipse Desktop was packaged as a JAR (a.k.a. Java ARchive). We reuse the M2T transformation engine packaged as a JAR in Eclipse Cloud. ArgonML provides an Ecore XMI file compatible with the M2T transformation engine.

3.3 Limitations

ARGON Cloud currently demonstrates four notable limitations. Firstly, ARGON Cloud depends on Infrastructure as Code (IaC) tools to orchestrate infrastructure provisioning on each cloud provider. While user feedback indicates that the ARGON Cloud is appropriate for modeling cloud infrastructure resources, they suggest a seamless integration of ARGON Cloud with IaC tools to avoid moving IaC scripts —generated by ARGON— toward IaC tools such as Ansible, Terraform, CloudFormation, etc.

The second limitation of ARGON Cloud is that it does not simultaneously support the modeling of infrastructure resources for multi-clouds. It is worth mentioning that ARGON has an agnostic infrastructure metamodel (IMM), which means its Domain-Specific Modeling Language (i.e., ArgonML) can model infrastructure resources for different cloud providers. However, ArgonML can model the infrastructure for one cloud provider at a time.

A third limitation is that ARGON Cloud does not support calculating the estimated cost of provisioning infrastructure resources. Since ARGON can model the infrastructure of different cloud providers, a significant limitation is that it does not calculate the estimated cost of the infrastructure modeled that will be provisioned in a particular cloud provider.

Finally, ARGON Cloud is working in an isolated fashion. On the one hand, ARGON Cloud models the infrastructure resources for several cloud providers. On the other, it generates IaC scripts for different provisioning tools. However, ARGON Cloud lacks a seamless integration with software development tools. to support an entire DevOps lifecycle. As a result, we have to develop the software and define the infrastructure in different environments.

4. RUNNING EXAMPLE

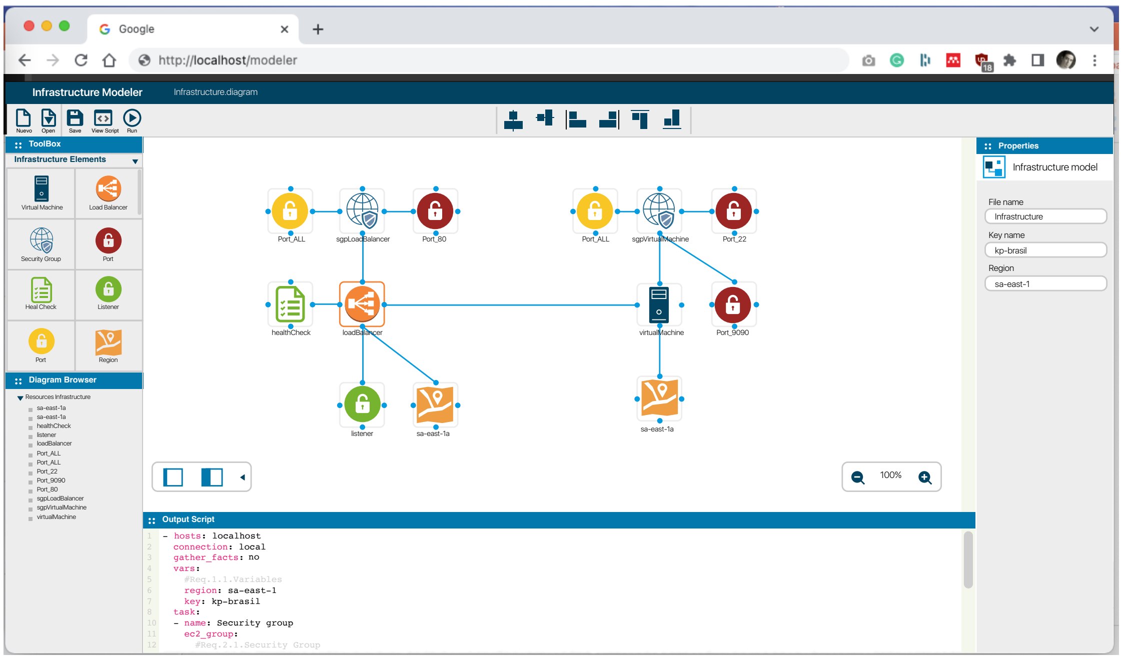
To demonstrate the feasibility of ARGON Cloud as a Software-as-a-Service approach, we propose a case study description to explain how to model and then generate the corresponding script for infrastructure provisioning in Amazon Web Services (AWS).
The infrastructure requirements for modeling and provisioning are the following:

- **Requirement 1.** A Load Balancer Architecture should define infrastructure resources needed for provisioning them in the Brazil region of AWS.
- **Requirement 2.** A *Load Balancer* element should distribute the workload among five (5) EC2 Instances (a.k.a. virtual machines). Each EC2 Instance should have one (1) virtual CPU and one (1) GB of RAM.
- **Requirement 3.** A *Health Check* element should review the state of each EC2 Instance through port 9090 and TCP protocol utilizing checking intervals every 24 seconds and should wait at least eight (8) seconds to notify an error. Moreover, the element should receive at least three (3) consecutive errors to change the state of an EC2 instance to an unhealthy state. In contrast, before changing the state of an EC2 Instance to a healthy state, it is necessary to obtain at least six (6) state verification probes successfully.
- **Requirement 4.** A *Listener* element should use the TCP protocol to resolve client requests through port 80 and distributes the workload to EC2 instances through port 9090.
- **Requirement 5.** All infrastructure resources should work in the first availability zone of the Brazil region in AWS.
- **Requirement 6.** A *Security Group* element should enable inbound TCP connections to the *Load Balancer* solely through port 80. Additionally, all outbound *Load Balancer* connections should be allowed.
- **Requirement7.** A different Security Group element should enable inbound TCP connections to the EC2 Instances through port 9090 and port 22. Additionally, all outbound connections from EC Instances should be allowed.

4.1 Cloud Infrastructure Modeling

Figure 6 shows an Infrastructure Diagram modeled by ARGON Cloud. Note that the resulting infrastructure model is the input for the M2T transformation engine that generates a script of infrastructure provisioning for Amazon Web Services.
The modeling solution for the infrastructure requirements is explained below:

- **Solution Requirement 1.** Figure 6 shows the Infrastructure Diagram properties that specify the **kp-brazil** code in the *Key name* property and the **sa-east-1** code in the *Region* property to provision the infrastructure in the Brazil region of AWS. The *File name* property is the name of the Infrastructure Diagram and the Script.
- **Solution Requirement 2.** Figure 8a shows the **Load Balancer** element, where it is necessary to fill out the *Name* property as well as the security group in the *Group* property and the EC2 Instance in the *Machines* property. Figure 7a presents the EC2 Instance properties. Note that the EC2 Instance defines several virtual machines to be provisioned with identical hardware characteristics. We specify the number **5** in the *Count* property to determine that five EC2 Instances will be running with the Load Balancer. The **ami-6d7t8ddee49d3b0f0** property defines the Image code to set up that each EC2 Instance should have the OS Ubuntu 18.04 LTS with the Apache server. The **t2.micro** code is specified in the *Instance type* property to set that each virtual machine should have 1 virtual CPU and 1 GB of RAM.
- **Solution Requirement 3.** Figure 8c shows the properties of the **Health Check** element. The *Ping protocol* property has the **TCP** option selected, and the *Ping port* property has the number **9090** to allow checking the state of each EC2 Instance through the **TCP** protocol and port **9090**. The *Interval* property has the number **24**, and the *Response timeout* property has the number **6** to enable checking intervals every **24** seconds and the timeout at least **6** seconds to notify an error. The *Healthy threshold* property has the number **8**, and the *Unhealthy threshold* property has the number **3** to configure the change of the state of an EC2 Instance. On the one hand, if the element receives at least **3** consecutive errors from an EC2 Instance, it should change the EC2 Instance state to unhealthy. On the other hand, if the element receives at least **8** successful consecutive state verification probes, it should change the EC2 Instance state to healthy.
- **Solution Requirement 4.** Figure 8b shows the properties of the **Listener** element. The *Protocol* property has the **TCP** option selected to enable the TCP protocol to resolve client requests. The *Load balancer port* property has the number **80**, and the *Instance port* property has the number **9090** to ensure client requests are resolved through port **80** and distributed to the EC2 Instances through port **9090**.
- **Solution Requirement 5.** Figure 7d shows the **Zone** element. The Load Balancer and Virtual Machine elements have a **Zone** element connected. To ensure that the Load Balancer and its EC2 Instances (i.e., virtual machines) work in the first availability zone of the Brazil region, we should fill out the **sa-east-1a** code in the *Name* property.
- **Solution R6.-** Figure 8d shows the **Inbound** element of the Security Group for the Load Balancer. The *Protocol* property has the **TCP** option selected, and the properties *From port* and *To port* have the number **80** to enable all inbound **TCP** connections to the Load Balancer to be made through port 80. Figure 7c shows the *Outbound* element, similar to the case of the Security Group for the Load Balancer. The *Outbound* element has the *Protocol* property with the **ALL** option selected to enable all outbound Load Balancer connections.
- **Solution R7.** Figure 7b shows the *Inbound* element of the Security Group for EC2 Instances. The *Protocol* property

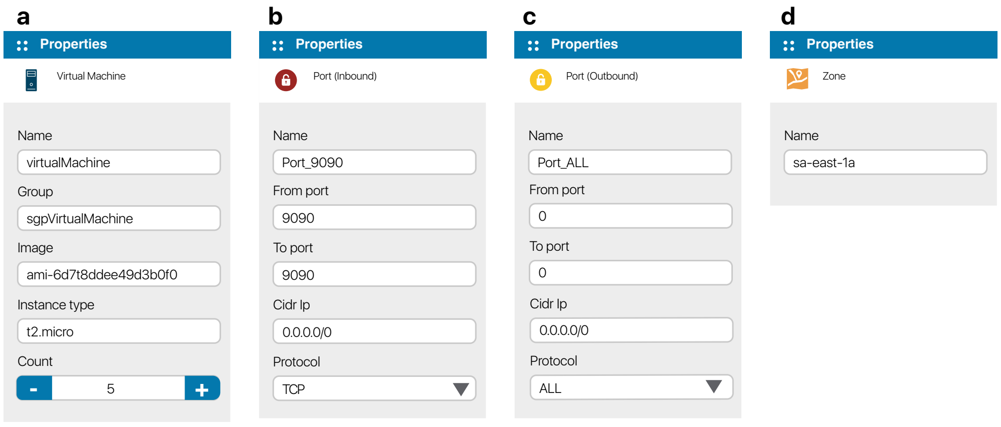


File name: Infrastructure
Key name: kp-brasil
Region: sa-east-1

Output Script

```
- hosts: localhost
  connection: local
  gather_facts: no
  vars:
    #Req.1.1.Variables
    region: sa-east-1
    key: kp-brasil
  task:
  - name: Security group
    ec2_group:
      #Req.2.1.Security Group
```

Figure 6. Infrastructure Diagram for Amazon Web Services



a — Virtual Machine

Name	virtualMachine
Group	sgpVirtualMachine
Image	ami-6d7t8ddee49d3b0f0
Instance type	t2.micro
Count	5

b — Port (Inbound)

Name	Port_9090
From port	9090
To port	9090
Cidr Ip	0.0.0.0/0
Protocol	TCP

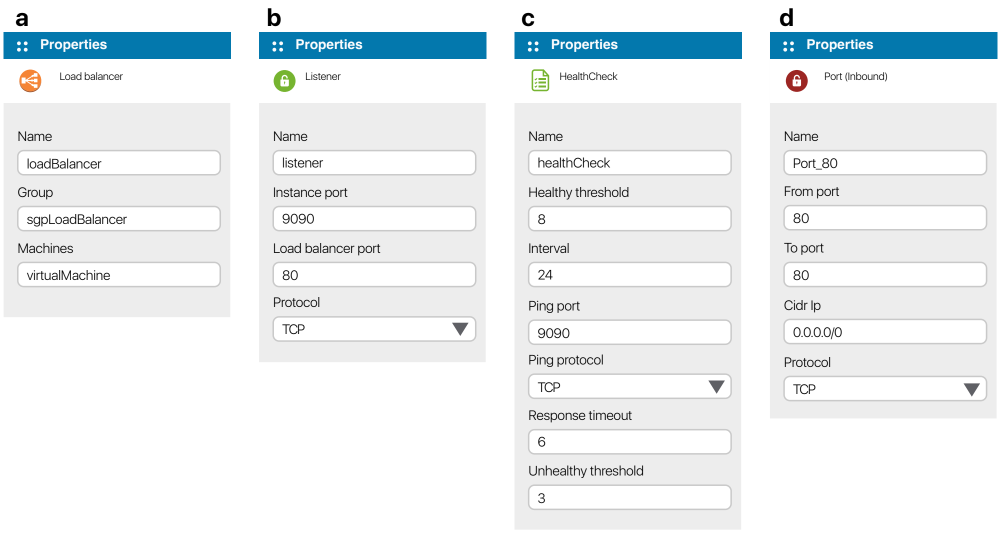
c — Port (Outbound)

Name	Port_ALL
From port	0
To port	0
Cidr Ip	0.0.0.0/0
Protocol	ALL

d — Zone

Name	sa-east-1a

Figure 7. Virtual Machines elements



a — Load balancer

Name	loadBalancer
Group	sgpLoadBalancer
Machines	virtualMachine

b — Listener

Name	listener
Instance port	9090
Load balancer port	80
Protocol	TCP

c — HealthCheck

Name	healthCheck
Healthy threshold	8
Interval	24
Ping port	9090
Ping protocol	TCP
Response timeout	6
Unhealthy threshold	3

d — Port (Inbound)

Name	Port_80
From port	80
To port	80
Cidr Ip	0.0.0.0/0
Protocol	TCP

Figure 8. Load Balancer elements

has the **TCP** option selected, and the properties *From port* and *To port* have the number **9090** to enable all inbound TCP connections to the Apache server to be made through port 9090. Another Inbound element has the Protocol property with the TCP option selected, the properties From port and To port have the number 22 to enable SSH connections to EC2 Instances through port 22. Finally, Figure 7c shows the *Outbound* element where the Protocol property has the **ALL** option selected to ensure that all outbound EC2 Instance connections are enabled.

4.2 Cloud Infrastructure Provisioning

ARGON Cloud allows modeling the cloud infrastructure provisioning for AWS on an infrastructure model and then generating the script for a particular DevOps provisioning tool, such as Ansible, Terraform, etc. For instance, Figure 9 shows an excerpt from the Ansible script generated from the infrastructure model (see Figure 6).

The Ansible script (see Figure 9) presents the Infrastructure Diagram properties, such as *region* (line 6) and *key* (line 7), to be used to provision the infrastructure in the Brazil region of AWS. In the task section, the security group for the EC2 Instances (see requirement R7) defines the *region* (line 11) in which they will run, along with its *name* (line 12) and a *description* (line 13). The security group inbound rules enable EC2 Instances to use TCP *protocol* to connect (lines 15 and 19) through *Port* 9090 (lines 16 and 17) and *Port* 22 (lines 20 and 21). Furthermore, the EC2 Instances specify the *region* (line 29) in which they will run, *kp_name* (line 30) is the key pair of access to the Brazil region, *instance_type* (line 31) defines the hardware characteristics (e.g., 1 vCPU and 1GB RAM), *image* (line 32) is the AMI (Amazon Machine Image) code provides the information required to launch an instance, *exact_count* (line 35) indicates how many instances will be launched, and *zone* (line 39) indicates availability zone in which to launch the virtual machines.

```
1   ---
2   - hosts: localhost
3     connection: local
4     gather_facts: no
5     vars:
6       region: sa-east-1
7       key: kp-brazil
8     task:
9     - name: Security Group sgpVirtualMachine
10      ec2_group:
11        region: sa-east-1
12        name: sgpVirtualMachine
13        description: Security Group sgpVirtualMachine
14        rules:
15        - proto: TCP
16          from_port: 9090
17          to_port: 9090
18          cidr_ip: 0.0.0.0/0
19        - proto: TCP
20          from_port: 22
21          to_port: 22
22          cidr_ip: 0.0.0.0/0
23        rules_egress:
24        - proto: all
25          cidr_ip: 0.0.0.0/0
26      register: sgpVirtualMachine
27    - name: EC2 Instances virtualMachine
28      ec2:
29        region: sa-east-1
30        key_name: kp-brazil
31        instance_type: t2.micro
32        image: ami-6d7t8ddee49d3b0f0
33        instance_tags:
34          Name: virtualMachine
35        exact_count: 5
36        count_tag:
37          Name: virtualMachine
38        group: sgpVirtualMachine
39        zone: sa-east-1a
40        wait: yes
41      register: virtualMachine
```

Figure 9. Excerpt from Ansible script

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented ARGON Cloud as a Software-as-a-Service (SaaS) approach. We demonstrated the feasibility of migrating ARGON from a desktop environment to a shared cloud service and made available as a software product that represents a typical profile of a SaaS service. On the one hand, we explained how a domain-specific modeling language (i.e., ArgonML) was developed using JavaScript Frameworks, such as Ecore.js and mxGraph. However, we used a Model-to-Text (M2T) transformation engine to generate IaC scripts. Further, we explained a REST web service to interchange data between ArgonML and the M2T transformation engine. Accordingly, we presented the practicability of ARGON Cloud, running an example through modeling infrastructure resources to Amazon Web Services (AWS) and then generating the corresponding Infrastructure as Code (IaC) script for the Ansible tool.

Our research achieves migration ARGON toward Cloud Computing to provide infrastructure modeling and script generation as a SaaS service. Nevertheless, there are limitations to the use of ARGON Cloud and how to improve its features. First, we need an IaC tool like Ansible or Terraform to execute the cloud infrastructure provisioning. Currently, ARGON Cloud allows modeling infrastructure resources and generates the corresponding IaC scripts. However, ARGON Cloud must integrate IaC tools and connections to several providers to provide seamless infrastructure provisioning to a particular provider. Second, ARGON Cloud does not simultaneously support the infrastructure modeling for multi-cloud providers. This might be a feature to improve because, as far as we know, no tool allows modeling multi-cloud infrastructure resources. Third, ARGON Cloud does not calculate the cost of provisioning infrastructure resources modeled. This feature is implemented for cloud providers like Amazon Web Services with AWS Pricing Calculator services. In this scenario, developing a pricing calculator in ARGON Cloud could help practitioners understand the cost of infrastructure modeled for a particular cloud provider. The challenge is to know the cost of each infrastructure resource of each cloud provider. Finally, ARGON Cloud works in an isolated fashion, which means it needs to be integrated with software development tools to achieve a holistic DevOps lifecycle.

In future work, we want to extend the ARGON Cloud features, as mentioned afore, such as i) integrating IaC tools to achieve our tool orchestrate de infrastructure provisioning; ii) modeling infrastructure resources for multi-cloud; iii) developing a pricing calculator of infrastructure resources to obtain the price of cloud infrastructure modeled; and iv) integrating software development tools to achieve a full DevOps lifecycle. We also plan to run experiments with practitioners and students with experience in software development and, in particular, with knowledge of cloud computing.

ACKNOWLEDGMENTS

This research work was supported by the Escuela Politécnica Nacional (Ecuador) under **PII-DICC-2021-01** and **PII-DICC-2021-02** projects.

REFERENCES

Brambilla, Marco, Jordi Cabot, and Manuel Wimmer (2017). *Model-Driven Software Engineering in Practice*. 2nd. Morgan and Claypool Publishers. ISBN: 9781627057080.

Brikman, Yevgeniy (2019). *Terraform: Up and Running*. O'Reilly Media, Inc., p. 368. ISBN: 9781492046905.

Buyya, Rajkumar, James Broberg, and Andrzej Goscinski (Jan. 2011). *Cloud Computing: Principles and Paradigms*. John Wiley and Sons. ISBN: 9780470887998.

Casola, Valentina et al. (Aug. 2017). «MUSA deployer: Deployment of Multi-cloud Applications». In: Institute of Electrical and Electronics Engineers Inc., pp. 107–112. ISBN: 9781538617588. DOI: https://doi.org/10.1109/WETICE.2017.46.

Chen, Wei et al. (Aug. 2016). «MORE: A model-driven operation service for cloud-based IT systems». In: Institute of Electrical and Electronics Engineers Inc., pp. 633–640. ISBN: 9781509026289. DOI: https://doi.org/10.1109/SCC.2016.88.

Chiari, Michele et al. (2022). «Developing a New DevOps Modelling Language to Support the Creation of Infrastructure as Code». In: *Communications in Computer and Information Science* 1617 CCIS, pp. 88–93. DOI: https://doi.org/10.1007/978-3-031-23298-5_8.

Farley, Dave and Jez Humble (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, p. 300. ISBN: 9780321670250.

Ferry, Nicolas et al. (Jan. 2018). «CloudMF: Model-Driven Management of Multi-Cloud Applications». In: *ACM Transactions on Internet Technology (TOIT)* 18 (2). DOI: https://doi.org/10.1145/3125621. URL: https://dl.acm.org/doi/10.1145/3125621.

Guerriero, Michele et al. (2019). «Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry». In: pp. 580–589. DOI: https://doi.org/10.1109/ICSME.2019.00092.

Kartheeyayini, V. et al. (2022). «AWS cloud computing platforms deployment of landing zone - Infrastructure as a code». In: vol. 2393. DOI: https://doi.org/10.1063/5.0079757.

López-Viana, Ramón, Jessica Díaz, and Jorge E. Pérez (2022). «Continuous Deployment in IoT Edge Computing: A GitOps implementation». In: *2022 17th Iberian Conference on Information Systems and Technologies (CISTI)*, pp. 1–6. DOI: https://doi.org/10.23919/CISTI54924.2022.9820108.

Miñón, Raúl et al. (2022). «Pangea: An MLOps Tool for Automatically Generating Infrastructure and Deploying Analytic Pipelines in Edge, Fog and Cloud Layers». In: *Sensors* 22(12). DOI: https://doi.org/10.3390/s22124425.

Morris, Kief (2016). *Infrastructure as Code : Managing Servers in the Cloud*. O'Reilly Media, Inc. ISBN: 9781491924358.

Neharika, Keerthi and Ruth G. Lennon (2023). «Investigations into Secure IaC Practices». In: *Proceedings of Seventh International Congress on Information and Communication Technology*. Ed. by Xin-She Yang et al. Springer Nature Singapore: Singapore, pp. 289–303. ISBN: 978-981-19-1610-6. DOI: https://doi.org/10.1007/978-981-19-1610-6_25.

Nitto, Elisabetta Di et al. (2017). *Model-Driven Development and Operation of Multi-Cloud Applications*. Ed. by Elisabetta Di Nitto et al. 1st. Springer Cham. DOI: https://doi.org/10.1007/978-3-319-46031-4.

Palma, Stefano Dalla, Dario Di Nucci, and Damian Tamburri (2022). «Defuse: A Data Annotator and Model Builder for Software Defect Prediction». In: *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 479–483. DOI: https://doi.org/10.1109/ICSME55016.2022.00063.

Rossini, Alessandro (2015). «Cloud Application Modelling and Execution Language (CAMEL) and the PaaSage Workflow». In: *Advances in Service-Oriented and Cloud Computing — Workshops of ESOCC 2015, Taormina, Italy*. ISBN: 978-3-319-33312-0. DOI: https://doi.org/10.1007/978-3-319-33313-7.

Sandobalin, Julio, Emilio Insfran, and Silvia Abrahao (2017a). «An Infrastructure Modelling Tool for Cloud Provisioning». In: *Proceedings of the IEEE 14th International Conference on Services Computing, SCC 2017*. ISBN: 9781538620052. DOI: https://doi.org/10.1109/SCC.2017.52.

Sandobalin, Julio, Emilio Insfran, and Silvia Abrahao (2017b). «End-to-end automation in cloud infrastructure provisioning». In: *Proceedings of the 26th International Conference on Information Systems Development, ISD 2017*. ISBN: 97899963228836. URL: http://aisel.aisnet.org/isd2014/proceedings2017/ISDMethodologies/5.

Sandobalin, Julio, Emilio Insfran, and Silvia Abrahao (2018). «An infrastructure modeling approach for multi-cloud provisioning». In: *Proceedings of the 27th International Conference on Information Systems Development: Designing Digitalization, ISD 2018*. ISBN: 978-91-7753-876-9. URL: http://aisel.aisnet.org/isd2014/proceedings2018/ISDMethodologies/2.

Sandobalin, Julio, Emilio Insfran, and Silvia Abrahao (2020). «On the effectiveness of tools to support infrastructure as code: Model-driven versus code-centric». In: *IEEE Access* 8. ISSN: 21693536. DOI: https://doi.org/10.1109/ACCESS.2020.2966597.

Solayman, Haleema Essa and Rawaa Putros Qasha (2023). «Seamless Integration of DevOps Tools for Provisioning Automation of the IoT Application on Multi-Infrastructures». In: *2023 3rd International Conference on Intelligent Communication and Computational Techniques (ICCT)*, pp. 1–7. DOI:

https://doi.org/10.1109/ICCT56969.2023.10075814.

Steinberg, David et al. (2009). *EMF: Eclipse Modeling Framework*. 2nd. Addison-Wesley Professional. ISBN: 0321331885.

Zhou, Huan et al. (2019). «CloudsStorm: A framework for seamlessly programming and controlling virtual infrastructure functions during the DevOps lifecycle of cloud applications». In: *Software - Practice and Experience 49*(10), pp. 1421–1447. DOI: https://doi.org/10.1002/spe.2741.

BIOGRAPHIES

Julio Sandobalín, is Professor at Escuela Politécnica Nacional, Ecuador. He received his Ph.D. in Computer Science and Master in Software Engineering from Universitat Politècnica de València, Spain. His research areas of interest are Model-Driven Engineering, Requirement Engineering, Empirical Software Engineering, DevOps, and Agile. Currently, his research focuses on the continuous delivery of cloud resources based on Model-Driven Engineering and DevOps

Carlos Iñiguez-Jarrín, is Professor in informatics at Escuela Politécnica Nacional (EPN), Ecuador. He has a Ph.D. in Computer Science from Universitat Politècnica de València (Spain), where he was a member of the Genomic Group at PROS Research Center. He holds a Master's Degree in Web Engineering from Universidad Politécnica de Madrid (Spain). He is currently involved in user experience design and interaction design.