End-to-End Automation in Cloud Infrastructure Provisioning

Julio Sandobalin *julio.sandobalin@epn.edu.ec*
Escuela Politécnica Nacional
Quito, Ecuador

Emilio Insfran *einsfran@dsic.upv.es*
Universitat Politècnica de València
Valencia, España

Silvia Abrahao *sabrahao@dsic.upv.es*
Universitat Politècnica de València
Valencia, España

Abstract

Infrastructure provisioning in the Cloud can be time-consuming and error-prone due to the manual process of building scripts. Configuration Management Tools (CMT) such as Ansible, Puppet or Chef use scripts to orchestrate the infrastructure provisioning and its configuration in the Cloud. Although CMTs have a high level of automation in the infrastructure provisioning, it still remains a challenge to automate an iterative development process based on models for infrastructure provisioning in the Cloud. Infrastructure as Code (IaC) is a process whereby the infrastructure is automatically built, managed, and provisioned by using scripts. However, there are several infrastructure provisioning tools and scripting languages that need to be used coherently. In previous work, we have introduced the ARGON modelling tool with the purpose of abstracting the complexity of working with different DevOps tools through a domain specific language. In this work, we present an end-to-end automation for a toolchain for infrastructure provisioning in the Cloud based on DevOps community tools and ARGON.

Keywords: Infrastructure as Code, cloud services, DevOps, Continuous Integration, Continuous Deployment, Continuous Delivery, Model-Driven Development.

1. Introduction

To succeed in a world where technologies, requirements, ideas, tools and timelines are constantly changing, information must be accurate, readily available, easily found and, ideally, constantly delivered in real-time to all team members [1]. In order to face these challenges, a new movement called DevOps [2] (Development & Operations) is promoting continuous collaboration between developers and operation staff through a set of principles and practices which optimize the delivery time of software, manage the Infrastructure as Code (IaC) and improve the user experience by basing on their continuous feedback. Infrastructure as Code [3] is an approach to infrastructure automation based on software development practices which emphasizes the use of consistent and repeatable routines for hardware provisioning. There exist a number of cloud-based DevOps processes which leverage services offered by Cloud Computing such as computing, networking, storage and elasticity. There are several DevOps community tools for infrastructure provisioning such as Ansible[1], Chef[2], Puppet[3] which use scripts for defining the final state of infrastructure provisioning in the

[1] https://www.ansible.com
[2] https://www.chef.io
[3] https://puppet.com

Cloud. However, practitioners are presently facing the challenge to manage different infrastructure provisioning tools, each one of them with their own scripting language. To mitigate this situation, in a previous work we introduced ARGON [4] (*An infRastructure modellinG tool for clOud provisioNing*), which aims to abstract the complexity of working with different DevOps provisioning tools through a Domain Specific Language (DSL). ARGON allows modelling an infrastructure model with the final state of hardware and software to be deployed in the Cloud. Furthermore, ARGON automatically generates scripts for different DevOps provisioning tools using model-to-text transformations.

In many of today's enterprises, one of the most important challenges is how to deliver a new idea or software artefact to customers as quickly as possible. In many software projects, releasing software applications is a manually intensive process and the environments that host the development and operation software are often crafted individually, usually by operation staff. Furthermore, there is a manual configuration management of production environments and deploying to production environment takes place only once development has been completed. Due to all these issues, infrastructure provisioning in the Cloud can be time-consuming and error-prone. To address these issues, we present an automated end-to-end toolchain for the development, version control, build, testing, and deployment of the infrastructure in the Cloud. This toolchain is based on DevOps community tools and ARGON. Our approach provides the necessary abstractions to deal with the complexity of using different DevOps tools to automate continuous delivery practices in Cloud provisioning. **Fig. 1** presents an overview of the infrastructure provisioning pipeline.

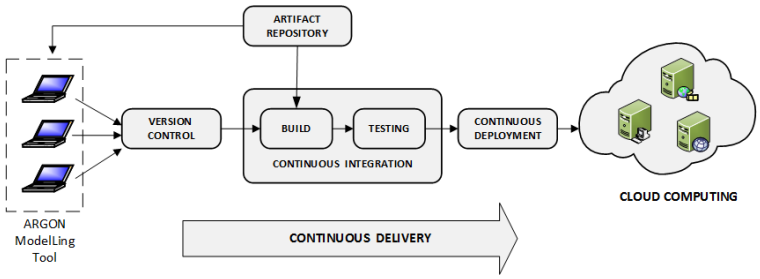

Fig. 1. Overview of the infrastructure provisioning pipeline.

We take advantage of the Infrastructure as Code concept in order to apply DevOps practices by supporting the automatic generation of scripts for managing the tools that are used for infrastructure provisioning in DevOps community. We use *ARGON Modeling Tool* in order to model the infrastructure provisioning and to obtain an infrastructure model. We model an infrastructure model from scratch or model an infrastructure architecture which is running in the Cloud. Subsequently, we will take this infrastructure model and push it towards a *version control* system. We use a version control system to retain and provide access to every version of every infrastructure model that has ever been stored on it. Moreover, this approach allows teams with infrastructure models across different places to work collaboratively.

Every infrastructure model must be checked into a single version control repository to begin the *Continuous Integration* stage. Continuous integration requires that every time some developer commits a change, the entire application be built and a comprehensive set of automated tests run against it [2]. We have developed a model-to-text transformation engine based on Acceleo[4] and Maven[5] to automatically *build* scripts from the infrastructure model. The model-to-text transformation engine is a JAR (Java ARchive) which is used as a plugin for a continuous integration server. An *artefact repository* is used to provide software libraries such as the text-to-model transformation engine and the ARGON's Domain Specific Language. We use an artefact repository in order to provide a unique provider of libraries or software artefact in stages of development, build, and testing. After the build stage, the scripts

[4] http://www.acceleo.org
[5] https://maven.apache.org

for infrastructure provisioning are ready to pass a set of automated *tests*. First, a syntax check test is executed to verify the structure of the scripts. Then, a static code test is performed by parsing code or definition files without executing them in the Cloud. Finally, an infrastructure test is implemented to ensure the correct operation of the infrastructure and applications deployed in the Cloud. Scripts that have been built and have overcome a set of automated tests are ready to be used in infrastructure provisioning tools. A *Continuous Deployment* stage takes these scripts built in the previous stage and automatically deploys them toward a Cloud platform. The main advantages of using an end-to-end automation for an effective toolchain for Cloud infrastructure provisioning are the following:

- The infrastructure can be easily created, destroyed, replaced, and resized
- Rebuilding any element of an infrastructure is possible with less effort in a reliable way
- It is possible to build identical infrastructure elements in different environments. This makes it possible to achieve the *parity* DevOps principle, which means having the same infrastructure in staging and production environments.
- It is possible to achieve the *reproducibility* DevOps principle due to the fact that any action that is carried out on the infrastructure should be repeatable
- It is possible to make frequent changes on the infrastructure model so the infrastructure in the Cloud can be safely and quickly modified

The remainder of this paper is structured as follows: Section 2 discusses related works and identifies the need of an end-to-end automation for an effective toolchain for cloud infrastructure provisioning in the Cloud. Section 3 introduces an illustrative case study. Section 4 presents the infrastructure provisioning pipeline. Finally, Section 5 presents our conclusions and future work.

2. Related Work

In recent years, there has been much interest on cloud-based DevOps research. The DevOps community has provided many Configuration Management Tools (CMT) for infrastructure provisioning such as Ansible, Puppet, Chef, among others. Despite the fact that CMTs have achieved the automation of the orchestration of infrastructure deployment and its configuration in the Cloud there are still many challenges to face.

TOSCA [5] is a standard for Topology and Orchestration Specification for Cloud Application which allows modelling nodes (virtual or physical machines) and orchestrates the deployment of Cloud applications. TOSCA uses DevOps provisioning tools such as Chef to infrastructure provisioning and Juju[6] for the deployment of cloud based applications. In [6] TOSCA has classified DevOps community tools into node-centric artefacts and environment-centric artefacts. The former are scripts that run on a server, virtual machines, or containers for infrastructure provisioning. The latter are scripts that run on multiple nodes and support the deployment of Cloud applications.

MORE [7] is a model-driven operation service for cloud-based IT systems that focuses on automating the initial deployment and the dynamic configuration of a system. MORE provides an online modelling environment to define a topology model to specify system structure and desired state. MORE transforms the topology model into executable code for the Puppet tool in order to get virtual machines, physical machines and containers.

MODAClouds [8] is a European project undertaken to simplify the Cloud service usage process. One of its goals is to deliver an Integrated Development Environment (IDE) to support system developers in building and deploying applications and related data to multi-Clouds spanning across the full Cloud stack. Energizer 4Clouds is an executable platform from MODAClouds which includes automatic infrastructure provisioning using specially-designed Puppet modules, the ability to use existing infrastructure and an API middleware for job control.

[6] https://jujucharms.com

Soni et al. [9] proposes a proof of concept for designing an effective framework for continuous integration, continuous testing, and continuous delivery to automate the source code compilation, code analysis, test execution, packaging, infrastructure provisioning, deployment and notifications using build pipeline concept. This approach focuses its efforts in necessities of the insurance industry to better respond to dynamic market requirements, faster time to market for new initiatives and services and support for innovative ways of customer interaction.

Rathod et al. [10] propose a framework for automated testing and deployment to help automated code analysis, test selection, test scheduling, environment provisioning, test execution, results from analysis and deployment pipeline. In test orchestration frameworks, it is typically very complicated to develop pipelines which make software reliable and bug-free.

An analysis of the works mentioned above shows that current approaches have focused their efforts in reusing the tools proposed by the DevOps community to solve gaps related to infrastructure provisioning and deployment of Cloud applications. However, as far as we know, there is no other previous approach to end-to-end automation for an effective toolchain of an infrastructure provisioning tool based on the Infrastructure as Code concept. Furthermore, we are taking advantage of the ARGON Modelling Tool to start an infrastructure delivery pipeline which is based on an infrastructure model.

3. Case Study Description

In order to illustrate the use of our approach, in this section we present an excerpt of a case study (adapted and extended from [4]). The case study is based on MOOC (Massive Open Online Courses). CEC University (CEC for short) offers massive and free courses which are accessible through the Internet. Over the last years, the demand for online courses has increased because MOOC have meant a revolution in the field of education, especially in universities. The main problem is the high demand for some courses which causes work overload in servers. In addition, students have difficulty in accessing video lessons and other multimedia materials. CEC has decided to solve this problem by purchasing new servers in order to create a cluster. However, new servers will not work when there is no demand for courses. To solve this dilemma, CEC has decided to migrate their infrastructure towards cloud computing and Amazon Web Services (AWS) has been selected as the Cloud platform. Furthermore, Ansible has been selected as the infrastructure provisioning tool attending to the fact that it does not require installing any agent like Chef or Puppet.

In order to resolve the problems mentioned above, the operation staff has decided to design two solutions. First, a scalable architecture that works on Cloud platforms for continuing enrolment courses. Second, an infrastructure provisioning of servers for courses with eventual enrolment. The first solution has been explained in detail in [4]. In this paper, we are focusing on the second solution due to the necessity to create specific infrastructure in the Cloud for every eventual course. The scope of this proposal is to create new EC2 instances in AWS where all the middleware, libraries and binaries files for software applications will be installed.

4. Infrastructure Provisioning Pipeline

4.1. ARGON Modelling Tool

ARGON is a modelling tool used for defining the final state of the infrastructure provisioning of Cloud resources and generating the scripts for the management of the provisioning tools used in the DevOps community. ARGON aims to abstract the complexity of working with different DevOps tools through a Domain Specific Language. ARGON allows modelling an infrastructure model and generate scripts for different DevOps provisioning tools. **Fig. 2** shows an infrastructure model which depicts the final state of infrastructure provisioning in AWS.

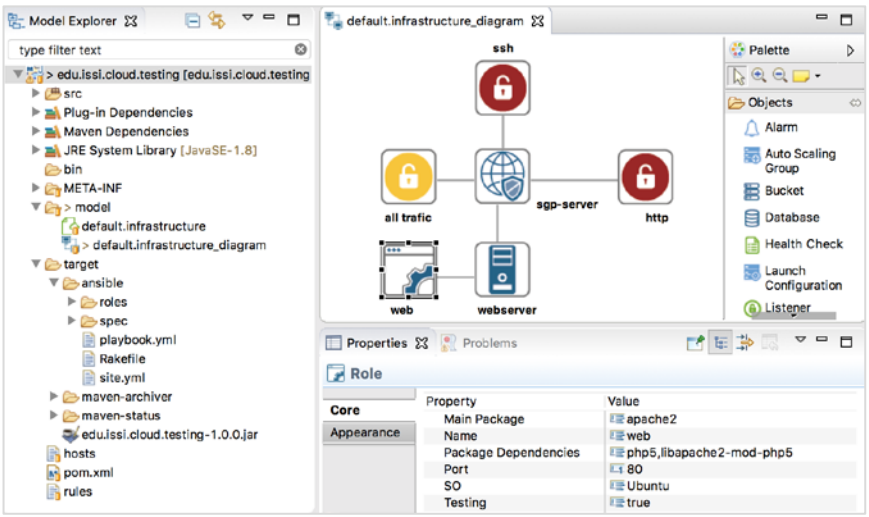

Fig. 2. Infrastructure Model defined using the ARGON tool.

The infrastructure model shows a security group (*sg-server*) which is like a firewall which enables connection of access ports to the EC2 instance (*webserver*). The security group has two inbound rules to allow access through port 22 for OpenSHH[7] connection (*ssh*) and port 80 for web applications connection (*http*). Moreover, an outbound rule (*all traffic*) is added to allow outgoing connections from EC2 instance. Additionally, a role (*web*) is linked to the EC2 instance in order to install an Apache web server (*apache2*) and their dependencies (*php5, libapache2-mod-php5*).

In the Attributes tab of *Role*, we can set up: *Main Package* with a web server, application server, database, etc. *Name* of Role. *Package Dependencies* which are all necessary software that the Main Package needs. *Port* is the port number that responds to user requests. *OS* stands for the operating system where the packages will be installed. *Testing* enables the execution of a set of automating tests. It is worth highlighting that every element in the infrastructure model has its own attributes. For example, Diagram attributes are: *File name* is script name, *Key name* is the name of the key pair file which is necessary to access AWS, *Region* is the region code where infrastructure will be deployed and *User* is the username which is necessary to install packages.

On the other hand, we are using Maven to develop the infrastructure project (*edu.issi.cloud.testing*) as it is a software project manager based on the concept of a project object model (POM). POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project. Furthermore, POM is very helpful in the case of collaborative development because it provides configuration details in a unique file for all team members.

```
12  <repositories>
13      <repository>
14          <id>nexus</id>
15          <url>http://nexusseerver:8081/nexus/content/groups/public/</url>
16      </repository>
17  </repositories>
18
19  <dependencies>
20      <dependency>
21          <groupId>edu.issi.cloud.ansible.playbook</groupId>
22          <artifactId>transformation-engine</artifactId>
23          <version>1.0.9</version>
24      </dependency>
25      <dependency>
26          <groupId>edu.issi.cloud</groupId>
27          <artifactId>dsl</artifactId>
28          <version>1.0.3</version>
29      </dependency>
30  </dependencies>
```

Fig. 3. Excerpt code of POM file of the infrastructure project.

[7] https://www.openssh.com

Fig. 3 shows an excerpt of the POM used in the infrastructure project. This excerpt presents the configuration of *artefact repository* (line 13-16) where *id* attribute is a unique identifier of the repository and *url* attribute is the reference to resources or dependencies. In the dependency section, we define the model-to-text transformation engine (line 22) and its version (line 23). Moreover, we define the Domain Specific Language DSL (line 27) and the version (line 28) used for the ARGON Modelling Tool.

4.2. Configuration Management

Configuration management refers to the process by which all artefacts relevant to a project, and the relationships between them, are stored, retrieved, uniquely identified and modified [2]. In order to get an effective toolchain for infrastructure provisioning, it is necessary to provide a configuration management strategy that determines how to manage all the changes that happen within the infrastructure project. It will also govern how team collaborates in a configuration management strategy.

Control Version System

A Control Version System is a mechanism for keeping multiple versions of files so that when a file is modified we can still access the previous revisions. Once the infrastructure model is ready, we can push it toward the Control Version System (see **Fig. 1**). We are using GitHub as a distributed Control Version System. GitHub has a *local repository* and a *remote repository* (see **Fig. 3**). The former gives each developer a local copy of the full development history. The latter imports all developed infrastructure projects and merges them into a unique infrastructure project. The basic workflow of GitHub (see **Fig. 3**) starts when a developer makes a *commit* of an infrastructure project towards the *local repository*. It is possible to make numerous *commits* towards the local *repository* and maintain a local history of the changes made to the infrastructure model. Therefore, when all team members decide to merge their models, it is necessary to make a *pull* of the infrastructure projects towards the *remote repository*. On the other hand, whenever a developer wishes to make changes to the infrastructure model stored in the *remote repository* it becomes necessary to make a *push* of the model towards the *local repository*. Therefore, to work with infrastructure project in ARGON becomes mandatory in order to make a *checkout* of the project from the *local repository*. Finally, the infrastructure project developed by ARGON is stored in GitHub (see **Fig.4**) and it is ready to move on to the Continuous Integration stage.

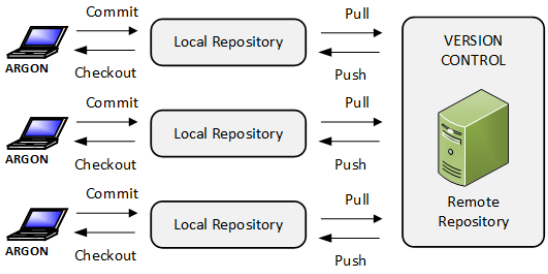

Fig. 3. Overview of Version Control with GitHub.

Fig. 4. Infrastructure project in GitHub.

Artifact Repository

Software libraries and files are managed differently due to the fact that libraries are typically deployed in the form of binary files or JAR (Java ARchive), they are never changed by the development team, and also because they are very rarely updated. Therefore, it becomes necessary to use an *Artefact Repository* for the management of a collection of JARs and their metadata. The artefacts or JARs will be used by clients such as Maven. We are using Sonatype Nexus[8] (Nexus for short) to manage the JARs used in the infrastructure project (see **Fig. 6**). ARGON requests artefacts to Nexus through a POM file (see **Fig. 3**) in order to resolve JAR dependencies and model the infrastructure project. However, ARGON also needs to resolve dependencies so as to compile with Maven and to bring forth the model-to-text transformation with Acceleo. In this case, Nexus works like a proxy providing artefacts from remote repositories such as Maven Central or Acceleo repository. As a result, Nexus provides artefacts for ARGON and *Continuous Integration Server* (see **Fig. 5**).

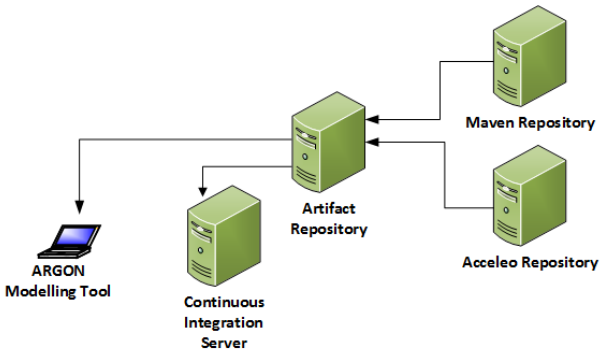

Fig. 5. Overview of Artifact Repository.

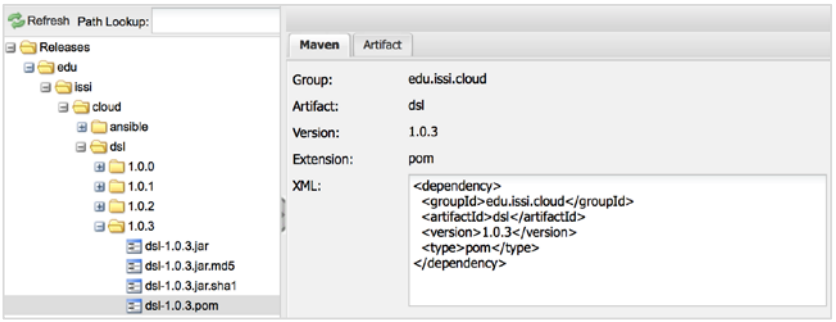

Fig. 6. Artifacts of the infrastructure project in Nexus.

4.3. Continuous Integration

Continuous integration requires that every time someone commits a change in the infrastructure model, the entire infrastructure project be built and a comprehensive set of automated tests run against the resulting scripts. For this reason, in the case of infrastructure provisioning, the goal of continuous integration is that the scripts stay in a deploying state all the time.

Continuous integration relies on certain prerequisites such as: 1) models in the infrastructure project must be checked into version control system and 2) the build process must be run in an automated way from a continuous integration environment. Therefore, the infrastructure project checked in GitHub is used as an input to the continuous integration stage and Jenkins[9] is accordingly used to run the build process in an automated way. Additionally, a comprehensive set of automated tests is run against resulting scripts. Finally,

[8] http://www.sonatype.org/nexus/
[9] https://jenkins.io

the result of the continuous integration pipeline (see **Fig. 7**) is a playbook or script for Ansible which is in a release state and it is input to the next continuous deployment stage.

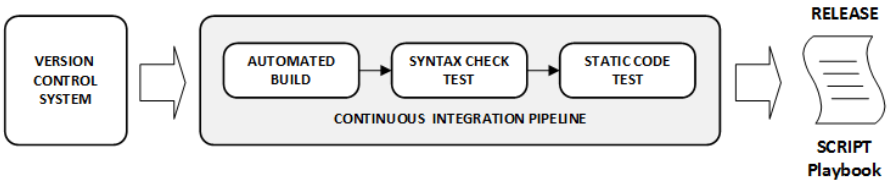

Fig. 7. Overview of Continuous Integration pipeline.

An Automated Build

We have developed a model-to-text transformation engine based on Maven that works in Jenkins and which is able to run a build process in an automated way. We have opted to use Jenkins as it is an application for continuous integration and continuous delivery of software projects. It is important to highlight that Jenkins can be run via the command line and this makes it possible to perform configuration based on Maven for the construction of scripts and run a set of automated tests over scripts and infrastructure deployed in the Cloud. In the POM file (see **Fig. 9**) of model-to-text transformation engine, we set up the java class (line 98) that should be used to register the package on which the model-to-text transformation is launched. Acceleo libraries (line 104-106) are configured to launch the transformation engine in the process-resources phase (line 109). In addition, the generator class must be specified (line 113) along with transformation rules, infrastructure model (line 114) as well as the folder where scripts are created (line 115). Finally, the infrastructure provisioning pipeline is created in Jenkins (see **Fig. 8**) where it shows that the last success pipeline created is number 72 with a time of 9 minutes 16 seconds. As it can be seen, the last pipeline failure is number 60 and this was 8 days ago.

Fig. 8. Dashboard of Jenkins.

```
97                <packagesToRegister>
98                    <packageToRegister>infrastructure.InfrastructurePackage</packageToRegister>
99                </packagesToRegister>
100           </configuration>
101       </plugin>
102
103       <plugin>
104           <groupId>org.eclipse.acceleo</groupId>
105           <artifactId>org.eclipse.acceleo.maven.launcher</artifactId>
106           <version>3.6.4</version>
107           <executions>
108               <execution>
109                   <phase>process-resources</phase>
110               </execution>
111           </executions>
112           <configuration>
113               <generatorClass>edu.issi.cloud.ansible.playbook.main.Generate</generatorClass>
114               <model>${model.infrastructure}</model>
115               <outputFolder>${folder.script}</outputFolder>
116           </configuration>
117       </plugin>
```

Fig. 9. Excerpt code of POM file of model-to-text transformation engine.

Automated Test Suite

The goal of automated testing is to help testing teams to keep the high quality of their systems by identifying errors as soon as they are produced so they can immediately be fixed [3]. During the software development process, there are three kinds of tests that must be in the running mode during the continuous integration stage: unit tests, component tests and acceptance tests. However, these tests cannot be applied to the infrastructure provisioning due to the fact that they belong to a different context. For this reason, we are following a set of automated testing for the infrastructure provisioning proposed in [3]. First, *Syntax Check Tests* are executed for the verification of the structure of the scripts: a) yamllint[10] is used for checking syntax validity of the scripts written in YAML[11] and cosmetic problems such as line length, trailing spaces, indentation, etc. b) Ansible is used to check the syntax of a playbook (scripts for Ansible). It uses ansible-playbook with the --syntax-check flag (see **Fig. 10**). This will run the playbook file through the parser to ensure its included files, the roles and other parameters have no syntax problems. Second, *Static Code Tests* are performed by parsing code or definition files without executing them in the Cloud. Ansible is later used to simulate the execution of a playbook: it uses ansible-playbook with the --check flag (see **Fig.11**). Ansible uses check mode to execute the static code tests. Check mode is just a simulation and will not make any changes to remote systems. Instead, any module instrumented to support check mode will report what changes would have taken place, rather than actually enabling them. However, in the case of the variables included in the playbook which have no value, they will be ignored. Finally, once the infrastructure provision is done, the *Infrastructure Tests* are executed towards ensuring the correct functioning of the infrastructure and the applications deployed in the Cloud. The infrastructure test will be tackled at the continuous deployment stage.

```
[infrastructure-provisioning-pipeline] $ /bin/sh -xe /tmp/hudson6922122130064775578.sh
+ cd edu.issi.cloud.testing/
+ ansible-playbook -i /etc/ansible/ec2.py --private-key /home/ubuntu/.ssh/kp-ireland.pem
target/ansible/playbook.yml --extra-vars pub_key_file=/home/ubuntu/.ssh/id_rsa.pub --syntax-check
```

Fig. 10. Syntax Check test for playbook.

```
playbook: target/ansible/site.yml
[infrastructure-provisioning-pipeline] $ /bin/sh -xe /tmp/hudson8602701116534883827.sh
+ cd edu.issi.cloud.testing/
+ ansible-playbook -i /etc/ansible/ec2.py --private-key /home/ubuntu/.ssh/kp-ireland.pem
target/ansible/playbook.yml --extra-vars pub_key_file=/home/ubuntu/.ssh/id_rsa.pub --check

PLAY [Infrastructure provisioning in the Cloud] ********************************

TASK [Provisioning Security Group(s)] ******************************************
changed: [localhost]

TASK [Provisioning EC2 instance(s)] ********************************************
skipping: [localhost]

TASK [Add new instance to host group] ******************************************
fatal: [localhost]: FAILED! => {"failed": true, "msg": "'dict object' has no attribute 'instances'"}
...ignoring

TASK [Waiting for the new instance to be ready] ********************************
fatal: [localhost]: FAILED! => {"failed": true, "msg": "'dict object' has no attribute 'instances'"}
...ignoring

PLAY RECAP *********************************************************************
localhost                  : ok=3    changed=1    unreachable=0    failed=0
```

Fig. 11. Static Check test for playbook.

[10] http://www.yamllint.com
[11] http://yaml.org

4.4. Continuous Deployment

Infrastructure deployment requires a series of steps such as configuring of tools, initializing data, configuring the infrastructure, the operating systems and the middleware. As projects get more complex, these steps become more numerous, longer, and more error-prone. Therefore, it is necessary to use tools which will perform infrastructure deployments in the Cloud. Configuration management tools such as Ansible or Puppet are typically used for automating the orchestration of infrastructure provisioning in the Cloud.

Continuous integration stage (see **Fig. 12**) starts once a playbook or script has overcome a set of automated tests and is ready to be used by Ansible. All commands to execute the playbook are set up in Jenkins. Firstly, the script for Ansible, called playbook.yml (see **Fig. 2**), is executed so as to get a virtual machine or EC2 instance (see **Fig. 13**) in Amazon Web Services. Once the EC2 instance named *webserver* is ready, the script called site.yml is executed in order to install all packages described in Role element (see **Fig. 2**).

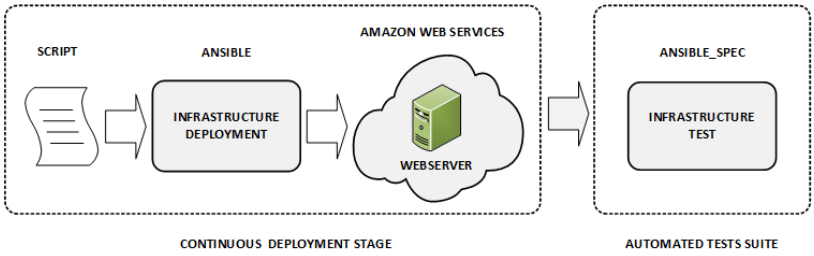

Fig. 12. Overview of Continuous Deployment stage.

Fig. 13. An EC2 instance (webserver) in Amazon Web Server.

Automated Test Suite

Once the scripts have successfully passed the automated tests suite at the continuous integration stage and each of them has done the infrastructure provisioning in the Cloud, the next step is to run *Infrastructure Tests* so as to verify that EC2 instance is running and packages installed. There are many tools for infrastructure testing: for example, Serverspec[12], RSpec[13] or Ansiblespec[14]. Serverspec is a tool that allows writing simple tests aimed at validating that a server is correctly configured. It can be used to remotely test the server state through an SSH connection. The development of Ansiblespec is based on Serverspec and contains some features of Ansible like dynamic inventory and roles. Ansiblespec is set up over Jenkins to validate that the EC2 instance is correctly configured and that the main package with its dependencies are correctly installed. ARGON allows specifying in every Role element (see **Fig. 2**) of the infrastructure diagram the creation of a set of tests. The model-to-text transformation engine creates the files needed for Ansiblespec. Given that the set of tests are created and executed automatically in Jenkins, we have defined four infrastructure tests: 1) Testing that the package is installed, 2) Testing that the package is

[12] http://serverspec.org
[13] http://rspec.info
[14] https://github.com/volanja/ansible_spec

enabled, 3) Testing that the package is running, and 4) The port of the package is listening. **Fig. 14** shows the console output of Jenkins with the executed infrastructure tests on the apache2 package. Finally, the result of testing is that apache2 is installed, enabled, running and port 80 is listening.

```
[infrastructure-provisioning-pipeline] $ /bin/sh -xe /tmp/hudson3827965938277251648.sh
+ cd edu.issi.cloud.testing/target/ansible/
+ /home/ubuntu/.rbenv/versions/2.2.3/bin/rake all
Run serverspec for webserver-TDD to {"uri"=>"52.17.231.186", "port"=>22}
/home/ubuntu/.rbenv/versions/2.2.3/bin/ruby -
I/home/ubuntu/.rbenv/versions/2.2.3/lib/ruby/gems/2.2.0/gems/rspec-support-
3.5.0/lib:/home/ubuntu/.rbenv/versions/2.2.3/lib/ruby/gems/2.2.0/gems/rspec-core-3.5.4/lib
/home/ubuntu/.rbenv/versions/2.2.3/lib/ruby/gems/2.2.0/gems/rspec-core-3.5.4/exe/rspec --pattern \{roles\}/\
{web\}/spec/\*_spec.rb

Package "apache2"
  should be installed

Service "apache2"
  should be enabled
  should be running

Port "80"
  should be listening

Finished in 3.72 seconds (files took 1.72 seconds to load)
4 examples, 0 failures
```

Fig. 14. Infrastructure Tests on EC2 instance.

4.5. Discussion

We have introduced a proposal for an end-to-end automation of infrastructure provisioning in the Cloud through a toolchain using DevOps community tools. There are several DevOps community tools[15] with which we can achieve other toolchains for infrastructure provisioning. However, we are using GitHub as it is a distributed control version system unlike Subversion or Mercurial. Moreover, GitHub provides a seamless connection with Jenkins and other integration servers. There exist several artefact repositories such as Nexus, Artifactory or Archiva among others. However, since we are using Maven in the creation of the infrastructure project and Maven works well with all the repositories mentioned above, we are using Nexus given the experience we have from previous projects. Similarly, there are alternatives to Maven like Gradle, Grunt or Rake, but we are using Maven as it is compatible with Acceleo and the Eclipse Modeling Framework [11], which is part of our model-to-text generation environment.

Jenkins is one of the most popular integration servers, although there are other tools like Bamboo, Travis CI, or Hudson. We could use Hudson or Travis CI to achieve the continuous integration stage, but we are using Jenkins due to our positive experience from previous projects. Ansible is a configuration management tool aimed at the orchestration of infrastructure provisioning which does not use any agent installed on the remote host, unlike Puppet or Chef. This is the main reason why we are using Ansible and also because it provides better control in an unattended installation.

Finally, we have showed an end-to-end automation for infrastructure provisioning in the Cloud. It is worth to mention that the we used ARGON [4] as a collaborative modelling tool in the infrastructure project. This approach may be useful for providing an automated infrastructure provisioning in research projects such as DIARy [12][13].

5. Conclusion and Future Work

In this paper, we have presented an end-to-end automation for an effective toolchain towards the development, version control, build, testing and deployment of infrastructure in the Cloud

[15] https://xebialabs.com/periodic-table-of-devops-tools/

by basing on DevOps community tools and ARGON. We have showed an effective automation of the Infrastructure as Code concept and how it can be implemented with the DevOps community tools. Although more validation is needed, we consider these results encouraging.

As future work, we want to extend the concept of Infrastructure as Code to the use of metrics for the monitoring of the toolchain of the infrastructure provisioning process and to get feedback which should lead to further improvement of the process. Primarily, we want to provide an automated infrastructure provisioning to software development projects towards a holistic DevOps solution. We also plan to run experiments with practitioners and students with experience in cloud computing development and, in particular, with knowledge in provisioning resources in the Cloud. This will help us to validate the effectiveness of the proposed solution for provisioning infrastructure on different platforms in the Cloud.

Acknowledgements: This research is supported by the Value@Cloud project (TIN2013-46300-R).

References

[1] C. A. Cois, J. Yankel, and A. Connell, "Modern DevOps: Optimizing software development through effective system interactions," in *IEEE International Professional Communication Conference*, 2015.

[2] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, 1 edition. Addison-Wesley Professional, 2010.

[3] K. Morris, *Infrastructure As Code: Managing Servers in the Cloud*, 1st ed. O'Reilly Media, Inc., 2016.

[4] J. Sandobalin, E. Insfran, and S. Abrahao, "An Infrastructure Modelling Tool for Cloud Provisioning," in *Proceedings - 14th IEEE International Conference on Services Computing (SCC)*, 2017.

[5] J. Wettinger, U. Breitenbücher, O. Kopp, and F. Leymann, "Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel," *Future Generation Computer Systems*, vol. 56, pp. 317–332, 2015.

[6] J. Wettinger, U. Breitenbucher, and F. Leymann, "Standards-based DevOps automation and integration using TOSCA," in *7th International Conference on Utility and Cloud Computing (UCC)*, pp. 59–68, 2014.

[7] W. Chen *et al.*, "MORE: A model-driven operation service for cloud-based IT systems," in *Proceedings - 13th IEEE International Conference on Services Computing, SCC*, pp. 633–640, 2016.

[8] E. Di Nitto, P. Matthews, D. Petcu, and A. Solberg, *Model-Driven Development and Operation of Multi-Cloud Applications*. Cham: Springer International Publishing, 2017.

[9] M. Soni, "End to End Automation on Cloud with Build Pipeline: The Case for DevOps in Insurance Industry, Continuous Integration, Continuous Testing, and Continuous Delivery," in *Proceedings - IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pp. 85–89, 2016.

[10] N. Rathod and A. Surve, "Test orchestration a framework for Continuous Integration and Continuous deployment," in *International Conference on Pervasive Computing: Advance Communication Technology and Application for Society (ICPC)*, 2015.

[11] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. 2008.

[12] M. Zuñiga-Prieto, S. Abrahao, and E. Insfran, "An Incremental and Model Driven Approach for the Dynamic Reconfiguration of Cloud Application Architectures," *24th International Conference on Information Systems Development (ISD)*. Harbin, China, 2015.

[13] M. Zúñiga-Prieto, E. Insfran, S. Abrahão, and C. Cano-Genoves, "Incremental Integration of Microservices in Cloud Applications," in *25th International Conference on Information Systems Development (ISD)*, 2016.