PIACERE



# Deliverable D3.4

# Infrastructural code generation - v1

| | |
|---|---|
| **Editor(s):** | Lorenzo Blasi |
| **Responsible Partner:** | HPE |
| **Status-Version:** | Version 1.0 – Final |
| **Date:** | 26.11.2021 |
| **Distribution level (CO, PU):** | PU |

| | |
|---|---|
| **Project Number:** | 101000162 |
| **Project Title:** | PIACERE |

| | |
|---|---|
| **Title of Deliverable:** | Infrastructural code generation - v1 |
| **Due Date of Delivery to the EC** | 30.11.2021 |

| | |
|---|---|
| **Workpackage responsible for the Deliverable:** | WP3 - Plan and create Infrastructure as Code |
| **Editor(s):** | Lorenzo Blasi (HPE) |
| **Contributor(s):** | Laurentiu Niculut (HPE CDS), Debora Benedetto (HPE CDS), Lorenzo Blasi (HPE) |
| **Reviewer(s):** | Radosław Piliszek (7BULLS)<br>Paweł Skrzypek (7BULLS) |
| **Approved by:** | All Partners |
| **Recommended/mandatory readers:** | WP3, WP4, WP5 |

| | |
|---|---|
| **Abstract**[1]**:** | These deliverables will present the outcome of Task T3.4. Each deliverable will comprise both a software prototype [KR3] and a Technical Specification Report. The document will include the ICG technical design and will report related research results. |
| **Keyword List:** | Code generation, Infrastructure as Code |
| **Licensing information:** | This work is licensed under Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) http://creativecommons.org/licenses/by-sa/3.0/ |
| **Disclaimer** | This document reflects only the author's views and neither Agency nor the Commission are responsible for any use that may be made of the information contained therein |



[1] This is the same deliverable description provided in the DoA

# Document Description

| Version | Date | Modifications Introduced | |
|---|---|---|---|
| | | Modification Reason | Modified by |
| v0.1 | 20.05.2021 | Definition of the ToC | Lorenzo Blasi, HPE |
| V0.2 | 04.11.2021 | First draft | Laurentiu Niculut, HPE CDS |
| V0.3 | 10.11.2021 | Second draft | Laurentiu Niculut, HPE CDS; Debora Benedetto, HPE CDS; Lorenzo Blasi, HPE |
| V0.4 | 11.11.2021 | Final editing. Version ready for review | Lorenzo Blasi, HPE |
| V0.5 | 25.11.2021 | Updated to answer reviewers' comments | Lorenzo Blasi, HPE |
| V0.9 | 26.11.2021 | Final Version accepted by internal reviewers | Lorenzo Blasi, HPE |
| V1.0 | 29.11.2021 | Ready for submission | Leire Orue-Echevarria, TECNALIA |

# Table of contents



# List of tables



# List of figures

# Terms and abbreviations

| | |
|---|---|
| AWS | Amazon Web Services |
| CSP | Cloud Service Provider |
| DevOps | Development and Operation |
| DoA | Description of Action |
| DOML | DevOps Modelling Language |
| EC | European Commission |
| GA | Grant Agreement to the project |
| GCP | Google Cloud Platform |
| IaC | Infrastructure as Code |
| ICG | Infrastructural Code Generator |
| IDE | Integrated Development Environment |
| IEM | IaC Execution Manager |
| IEP | IaC execution platform |
| IOP | IaC Optimization |
| IR | Intermediate Representation |
| JSON | JavaScript Object Notation |
| KPI | Key Performance Indicator |
| SW | Software |
| TBCG | Template-Based Code Generation |
| UML | Unified Modeling Language |
| VT | Verification Tool |

# Executive Summary

This deliverable describes the first implementation of the PIACERE Infrastructure Code Generator (ICG), which is the main output of the PIACERE Task 3.4. The state-of-the-art related to this component has been reported in section 2.3 (Generating IaC code from a model) of the PIACERE D3.1 deliverable.

This first prototype is more focused on the code generation functionalities than on the parsing of the input DevOps Modelling Language (DOML). This is because Task 3.4 adopted a bottom-up approach: instead of starting from the high-level input language (DOML), we started to analyse which are the major DOML concepts and the possible operations that DOML would require to be implemented in the generated Infrastructure as Code (IaC) language. Then we started developing IaC code for those concepts and operations, also referring to a specific example proposed in WP3 conference calls (the WordPress example reported in section 2.1.2). The developed IaC code evolved into templates and we identified which parameters were needed for each template to work: this was useful feedback that Task 3.4 provided to Task 3.2. At the end of this first year, some templates have been created for the major DOML concepts, such as VM, Network, Database, and others, along with a prototype of ICG that uses them to generate both Ansible and Terraform code. In the bottom-up approach, the ICG component closer to the DOML language, i.e. the Parser, has been left at the end and some experiments have been done to evaluate a possible implementation technology, as reported in section 2.1.2. The current ICG prototype contains only the backend part of the compiler, which generates IaC code from the developed templates, using as input a preliminary, handcrafted, version of the Intermediate Representation that will be created by the Parser.

The document starts by presenting, in section 2.1, the ICG internal architecture and describing each ICG component. The same section then lists the main functionalities planned for the ICG tool and identifies the relationship between those functionalities and the requirements collected in deliverable D2.1 and referred to the ICG. After describing how the ICG fits into the overall PIACERE architecture (section 2.1.1), the document reports about the experiment done to evaluate a possible implementation of the Parser component (section 2.1.2).

Section 2.2 then offers more technical details on each internal component of the ICG, from the Controller, managing the internal ICG workflow, to the Code Generator Plug-ins that, based on the available templates, generate the output Infrastructural code in the target language.

The following section 3 describes the structure of the released software and the content of the listed files, especially the Infrastructure as Code (IaC) templates, on which the code generation is based, are listed and described. The same section then explains how the released software can be installed and used.

Finally, section 4 reports the conclusions and indicates future steps. In particular, future releases will complete the planned functionalities and will improve several aspects of the ICG. For example, the Parser will be developed and a stricter relationship with the DOML will be defined; the Intermediate Representation will be enriched; internal components' interfaces will be cleaned up and documented, to improve ICG extensibility; and additional templates will be added.

# 1 Introduction

## 1.1 About this deliverable

This deliverable focuses on the structure and functionalities of the Infrastructural Code Generator (ICG) component of PIACERE. The document reports about what has been implemented so far in the lifecycle of the project, how it fits into the overall PIACERE framework, and how it can be installed and used. The state-of-the-art related to this component has been reported in section 2.3 (Generating IaC code from a model) of the PIACERE D3.1 deliverable. The plan for developing ICG is to have a first working prototype at M12 (this release); the next release (M24) will consolidate the code and add more features, taking especially into account both the Use Case requirements and the feedbacks from the integration phase; the last release (M30) will improve the component further, also focusing on its possible exploitation.

The ICG component will get DOML models in input and generate Infrastructure as Code (IaC) files as output, in languages such as Terraform[2] and Ansible[3]. The code-generation strategy adopted in the ICG component is called in the literature Template-Based Code Generation (TBCG) [1]. This technique allows producing code from partially complete code snippets, called templates. A template usually contains a fixed part, with some code in the target language that will be copied as-is into the output, and a variable part, with placeholders to be substituted with values from input parameters or even control statements to drive the generation process.

In this first year of the project, when we started working on ICG, the DOML language was not yet defined, se the decision has been to adopt a bottom-up approach: instead of starting from the high-level input language (DOML), we started to analyse which are the major DOML concepts and the possible operations that DOML would require to be implemented in the generated Infrastructure as Code (IaC) language. Then we started developing IaC code for those concepts and operations, also referring to a specific example proposed in WP3 conference calls (the WordPress[4] example reported in section 2.1.2). The developed IaC code evolved into templates and we identified which parameters were needed for each template to work: this was useful feedback that Task 3.4 provided to Task 3.2 for a more detailed definition of DOML. At the end of this first year, some templates have been created for the major DOML concepts, such as VM, Network, Database, and others, along with a prototype of ICG that uses them to generate both Ansible and Terraform code. In the bottom-up approach, the ICG component closer to the DOML language, i.e. the Parser, has been left at the end and only some experiments have been done to evaluate a possible implementation technology, as reported in section 2.1.2. The current ICG prototype contains only the backend part of the compiler, which generates IaC code from the developed templates, using as input a preliminary, handcrafted, version of the Intermediate Representation that will be created by the Parser.

## 1.2 Document structure

Section 2 of the document describes the functionalities and the technical specifications of the PIACERE ICG and explains what has been provided in this first prototype.
The D3.4 deliverable is a software deliverable, therefore this document also provides details about the released software in section 3: how it is structured, how it can be installed and used. In section 4 there are the conclusions and some indications about the next steps.

[2] https://github.com/hashicorp/terraform
[3] https://github.com/ansible/ansible
[4] https://wordpress.org/

# 2 Implementation

## 2.1 Functional description

This deliverable reports about the first implementation of the Infrastructural Code Generator (ICG) component of PIACERE.

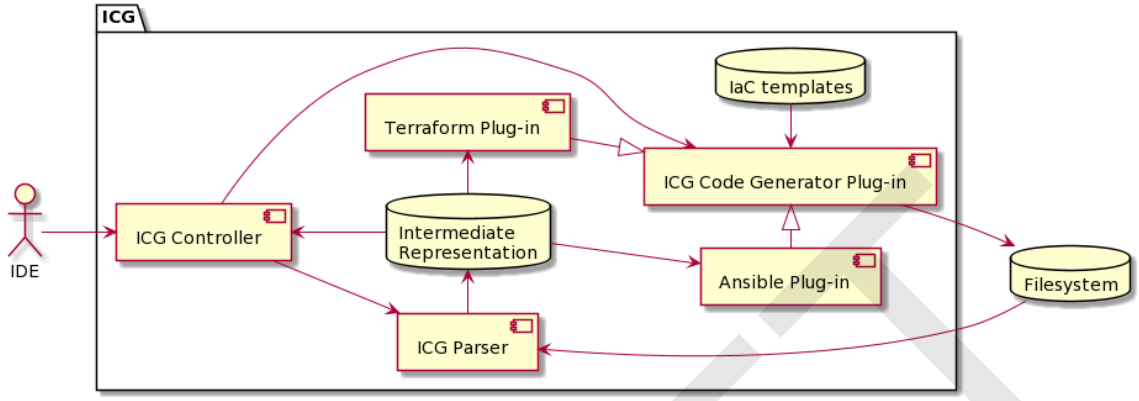


*Figure 1: ICG component representation*

The ICG, which is a command line application, takes as an input the DOML model produced by the IDE and oversees the generation of the Infrastructure as Code (IaC) files necessary for the deployment, configuration and orchestration of the represented user application and the needed infrastructure.

Figure 1 shows the internal ICG architecture, which includes the following components: ICG Controller, ICG DOML Parser, Intermediate Representation, ICG Code Generator Plug-in, and IaC templates.

**ICG Controller** is the main component that is started by the user as a command line tool, it reads and interprets the command line parameters and controls the internal flow of the other components.

**ICG DOML Parser** will be activated by the Controller to parse the input DOML and to produce the Intermediate Representation. Due to the unavailability of a clear DOML syntax definition[5], the Parser is not available in this first prototype. Nonetheless, we made some experiments, based on our assumptions, for the Parser component implementation. The experiments are reported in section 2.1.2.

The **Intermediate Representation** is created by the Parser and used both to drive the selection of the right Code Generator to use and to provide input to the Generator itself. More details on the Intermediate Representation are presented in section 2.2.2.3.

**ICG Code Generator** is the generic component, activated by the Controller, that reads the Intermediate Representation and, based on the available IaC templates, generates the output code in the selected language. The Code Generator can be considered a plug-in in the ICG architecture, and a specific Code Generator is expected to be available in the ICG for each target language. In this prototype a first release of the **Ansible Plug-in** and **Terraform Plug-in** are available, but more plug-ins may be added in the future to generate IaC code in other languages, for example to satisfy the Use Case requirements. The selected code generation strategy is called in the literature Template-Based Code Generation (TBCG) [1]. Each Code Generator plug-in

[5] DOML is reported in D3.1 and is being developed in parallel to the ICG.

selects the right templates, depending on the information read from the Intermediate Representation, and substitutes in these templates the values read from the Intermediate Representation itself.

The **IaC templates** are a collection of templates used by the Code Generator plug-ins to generate the target IaC code. As already indicated in the Introduction, templates usually contain a fixed part, with code in the target language to be copied as-is into the generated output, and a variable part, with placeholders to be substituted with values from input parameters. Multiple templates for each target language will be available: at least one for each supported DOML resource. A list of the templates developed for this first release can be found in section 3.1. Template examples are shown in section 2.2.2.4. In the future releases, implementing ICG extensibility, we will provide guidelines for writing new templates, so that expert users will be able to develop their own templates or to modify existing ones, both for supporting new DOML concepts and for providing support for new IaC languages.

The main functionalities planned for the Infrastructural Code Generator are the following.

F1. Read the input DOML model to extract all the needed information.

F2. Generate executable code for selected IaC languages.

F3. Provide enough extensibility to support the DOML extension mechanism [KR4]

F4. Provide enough extensibility to generate code for new IaC languages

F5. Generate IaC code that supports different cloud platforms

The listed functionalities will be implemented using an incremental approach over multiple releases. In this first release the following functionalities are implemented: **F2**, **F5**. Other functionalities are partially implemented.

The following Table 1 details the relationship between all the requirements indicated in deliverable D2.1 as related to ICG, and the ICG functionalities, with a description of the current coverage for each functionality.

*Table 1: Relationship between ICG functionalities and ICG requirements*

| Functionality | Req. ID | Requirement Description | Coverage |
|---|---|---|---|
| F1 | REQ96 | ICG must be able to read DOML language. | None. Since a parsable syntax for the DOML language is not yet completely defined, only an experiment has been done in this release, as reported in section 2.1.2. |
| F2 | REQ31 | ICG should provide verifiable and executable IaC generated from DOML for selected IaC languages (e.g., TOSCA/Ansible/Terraform). | An Intermediate Representation has been created by hand and Code Generation components have been implemented that read it and generate both Ansible and Terraform code. Currently Terraform code is generated for the Provisioning activity and Ansible code for the Configuration activity. |
| | REQ77 | ICG may generate IAC code for different supported/target tools according to the required | |

| Functionality | Req. ID | Requirement Description | Coverage |
|---|---|---|---|
| | | DevOps activity (as listed in REQ76). | |
| F3 | REQ41 | The IDE should be extensible through the plugin mechanism. Not only to support PIACERE assets (ICG, VT) but also for third party collaborators. | Partial. Even if the DOML Extension mechanism is not yet completely defined, the proposed architecture, based on plugins, and the selected code generation strategy, based on templates, allow the ICG to be extended to support new DOML concepts. |
| F4 | REQ41 | The IDE should be extensible through the plugin mechanism. Not only to support PIACERE assets (ICG, VT) but also for third party collaborators. | Partial. The proposed architecture, based on plugins, and the selected code generation strategy, based on templates, allow the ICG to be extended to support new IaC languages. |
| | REQ31 | ICG should provide verifiable and executable IaC generated from DOML for selected IaC languages (e.g., TOSCA/Ansible/Terraform). | |
| F5 | REQ29 | DOML should support the modelling of VM provisioning for different platforms such as (OpenStack, AWS) for canary and production environments. | The released templates supports the major cloud platforms: AWS.[6], Azure[7], GCP[8]. Templates for other platforms may be added in future releases, depending on Use Case requirements |

### 2.1.1 Fitting into overall PIACERE Architecture

The ICG is the PIACERE component tasked with the generation of the IaC files and as such will be integrated with the IDE, using the IDE integration facilities available for any compiler. Other PIACERE tools, such as the IaC Execution Manager (IEM) and the Verification Tool (VT) will access the IaC code generated by ICG to perform their tasked activities.

From the IDE it will receive the DOML model produced by the user, possibly verified by the Model Checker component. The model is used as an input by the ICG and contains the specifications of the desired infrastructure and configuration.

The VT will verify the generated IaC code, especially to evaluate its security.

The IEM will execute the IaC code produced by the ICG, to provision and configure the desired infrastructure.

[6] https://aws.amazon.com/
[7] https://azure.microsoft.com/
[8] https://cloud.google.com/

ICG is part of the PIACERE workflow executed at Design Time, as shown in the following Figure 2, which is explained in more detail in deliverable D2.1.

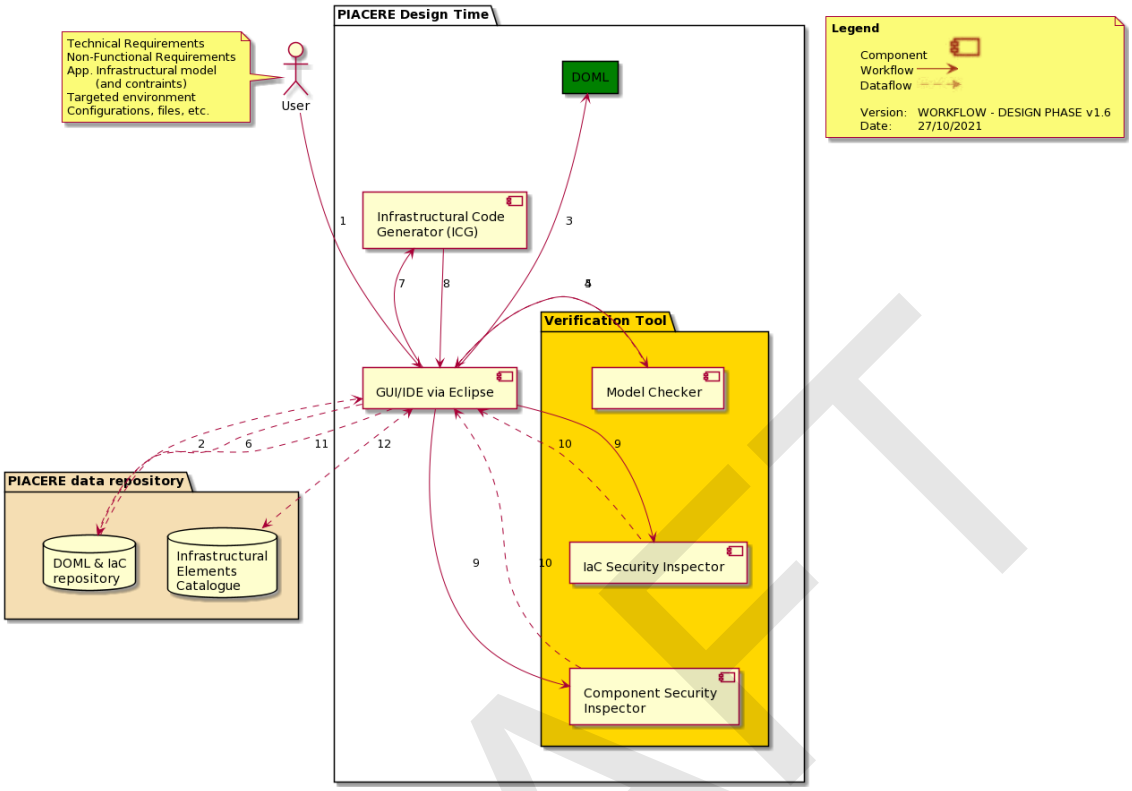


*Figure 2: PIACERE Design Time*

### 2.1.2 Parser experiments

The ICG DOML Parser goal is to extract information from the DOML in order to provide a suitable intermediate representation for the ICG Code Generator Plug-In. It is an intermediary component that decouples the DOML from the rest of the ICG components: if a modification on the DOML language occurred, the ICG DOML Parser will generate the agreed intermediate representation and the ICG Plugins will not have to be modified. Moreover, if a new plugin is implemented that requires an updated intermediate representation, the DOML will not be affected.

According to the requirements emerged from the study of the IaC files, we have identified the information to extract from the DOML based on the following considerations:

- There are different software tools that can be used to provide and configure the application environment, some of the most popular are Terraform for the creation of the infrastructure elements and Ansible for the automation of the application deployment and management.
- There are different cloud providers, such as AWS, Google Cloud or Microsoft Azure, each one with different infrastructure elements and configuration.
- There is custom information that the user must specify for the software installation. For example, for the installation of a database, the user will have to specify username, password and database name for the configuration to work.

The very first representation of the DOML that has been available was based on a UML diagram and the challenge was to translate this diagram to text files, so we selected Acceleo for a first experimental implementation of the ICG DOML Parser.

Acceleo (https://www.eclipse.org/acceleo/) is an open-source template-based source code generation technology integrated in the Eclipse IDE, suitable for parsing the DOML and transform it into files for any kind of language. Acceleo navigates the model, extracts information from it and produces the intermediate representation files for the other ICG modules.

The ICG DOML Parser experiment started considering a simple example representing a WordPress application (Figure 3). The application environment is made up by two virtual machines, one for the database installation and the other for the WordPress application.

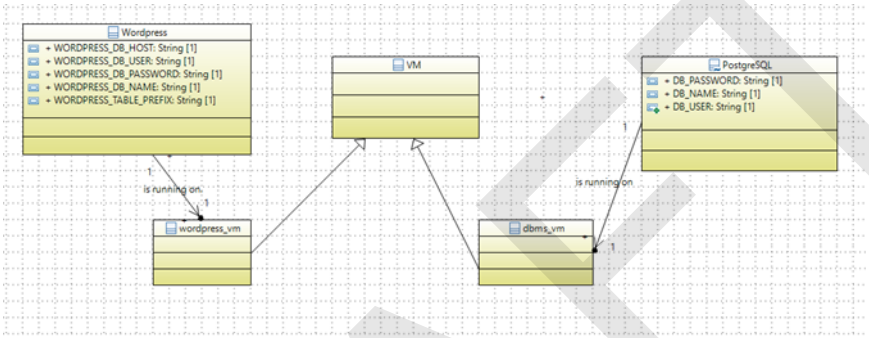


*Figure 3: WordPress UML diagram*

The model of the WordPress application is represented as an UML diagram and is parsed and navigated by the ICG DOML Parser using Acceleo. The output of the Parser is the *intermediate_representation.json* file containing the information needed by the ICG Plugins for the IaC file generation.

The main file of the Parser is *generate.mtl,* shown in Figure 4, which navigates the UML diagram, extracts information and transform it to the intermediate representation json file. Every time the generate.mtl file runs, the output file is overwritten.

```
[template public generateElement(model : Model)]
[comment @main/]
[file ('intermediate_representation.json', false, 'UTF-8')]
{
    "steps" : ['['/]
        {
            [generateTerraformParams(model.eContents(Class))/]
        },
        [for (a: Association | model.eContents(Association))]
        [generateAnsibleParams(a)/],
        [/for]
        ]
}
[/file]
[/template]
```

*Figure 4: Acceleo generate.mtl main file*

First, during the navigation of the UML diagram, the parser distinguishes the infrastructural elements from the software elements, in order to select the right software tool to deploy them.

In this example the virtual machines are infrastructural elements and the *generateTerraformParams()* function will take care about them, instead the WordPress application and the database are software elements and the *generateAnsibleParams()* will deal with them.

The *generateTerraformParams()* selects all the classes that generalize the VM class, in this example *wordpress_vm* and the *dbms_vm*, and write into the output JSON file the instructions for their deployment. The information extracted concerns:

- the software tool to use, that is Terraform
- the provider chosen, the default one is AWS
- the name of the virtual machines, that is the same as the ones on the UML diagram classes
- the id of each virtual machine
- RAM and CPU details, in this case the default values are used.

```
[template public generateTerraformParams(infrastructureClasses: Sequence(Class)) {virtualMachineId: Integer = 0;}]
"programing_lenguage": "terraform",
"output_path": "Output-code/",
"data":{
    "provider": "aws",
    [for (c:Class | infrastructureClasses)]
    [if (not c.generalization.general.name.contains('VM') -> isEmpty())]
    [virtualMachineId = virtualMachineId + 1 /]
    [generateTerraformVms(c, virtualMachineId)/],
    [/if]
    [/for]
}
[/template]

[template public generateTerraformVms(c: Class, virtualMachineId: Integer)]
"[c.name/]": ['['/]
    {
        "id": [virtualMachineId/],
        "ram": 128,
        "cpu": 32
    }
]
[/template]
```

*Figure 5: Terraform params generation*

These virtual machines will be the infrastructure on which will run the WordPress application and the database. The *generateAnsibleParams()* function adds to the JSON file the information for these three elements:

- the name of the virtual machine they are running on
- the custom parameters for the installation, such as the database username and password and a placeholder for their future values
- the folder in which to store the IaC files.

```
[template public generateAnsibleParams(a: Association)]
{
    "programing_lenguage": "ansible",
    "running_on": "[a.ownedEnd.getOtherEnd().type.name/]",
    "name": "[a.ownedEnd.type.name/]",
    "output_path": "Output-code/",
    "data":{
        "vars":{
            [for (params: Property | a.ownedEnd.type.eContents(Property))]
            [if (not params.name.size().oclIsInvalid())]
            "[params.name/]": "<<[params.name/]>>",
            [/if]
            [/for]
        },
        "play":{
            "OS": "debian"
        }
    }
}
[/template]
```

*Figure 6: Ansible params generation*

Finally, when all the classes are considered and all the information are extracted, the Parser produces the intermediate representation JSON file and stores it into the target folder (Figure 7). Next, the other component of the ICG module will read this file and use the instruction for the IaC files generation.

```
{
    "steps" : [
        {
            "programing_lenguage": "terraform",
            "output_path": "Output-code/",
            "data":{
                "provider": "aws",
                "wordpress_vm": [
                    {
                        "id": 0,
                        "ram": 128,
                        "cpu": 32
                    }
                ],
                "dbms_vm": [
                    {
                        "id": 1,
                        "ram": 128,
                        "cpu": 32
                    }
                ]
            }
        },
        {
            "programing_lenguage": "ansible",
            "running_on": "dbms_vm",
            "name": "PostgreSQL",
            "output_path": "Output-code/",
            "data":{
                "vars":{
                    "DB_PASSWORD": "<<DB_PASSWORD>>",
                    "DB_NAME": "<<DB_NAME>>",
                    "DB_USER": "<<DB_USER>>"
                },
                "play":{
                    "OS": "debian"
                }
            }
        },
        {
            "programing_lenguage": "ansible",
            "running_on": "wordpress_vm",
            "name": "Wordpress",
            "output_path": "Output-code/",
            "data":{
                "vars":{
                    "WORDPRESS_DB_HOST": "<<WORDPRESS_DB_HOST>>",
                    "WORDPRESS_DB_USER": "<<WORDPRESS_DB_USER>>",
                    "WORDPRESS_DB_PASSWORD": "<<WORDPRESS_DB_PASSWORD>>",
                    "WORDPRESS_DB_NAME": "<<WORDPRESS_DB_NAME>>",
                    "WORDPRESS_TABLE_PREFIX": "<<WORDPRESS_TABLE_PREFIX>>"
                },
                "play":{
                    "OS": "debian"
                }
            }
        }
        ]
}
```

*Figure 7: intermediate_representation.json*

The experiment illustrated above shows that Acceleo is a powerful tool for the transformation of the UML diagram in one or more files of any kind of type, it could easily translate the user model definition for an application into a textual format with structural instructions for its creation. The Parser implementation will still evolve and move forward following the improvement of the DOML, it will transform and provide all the data to the other ICG modules in the intermediate representation for the generation of the IaC files.

## 2.2 Technical description

### 2.2.1 Prototype architecture

The sequence diagram representing the internal flow of the ICG is shown in Figure 8.

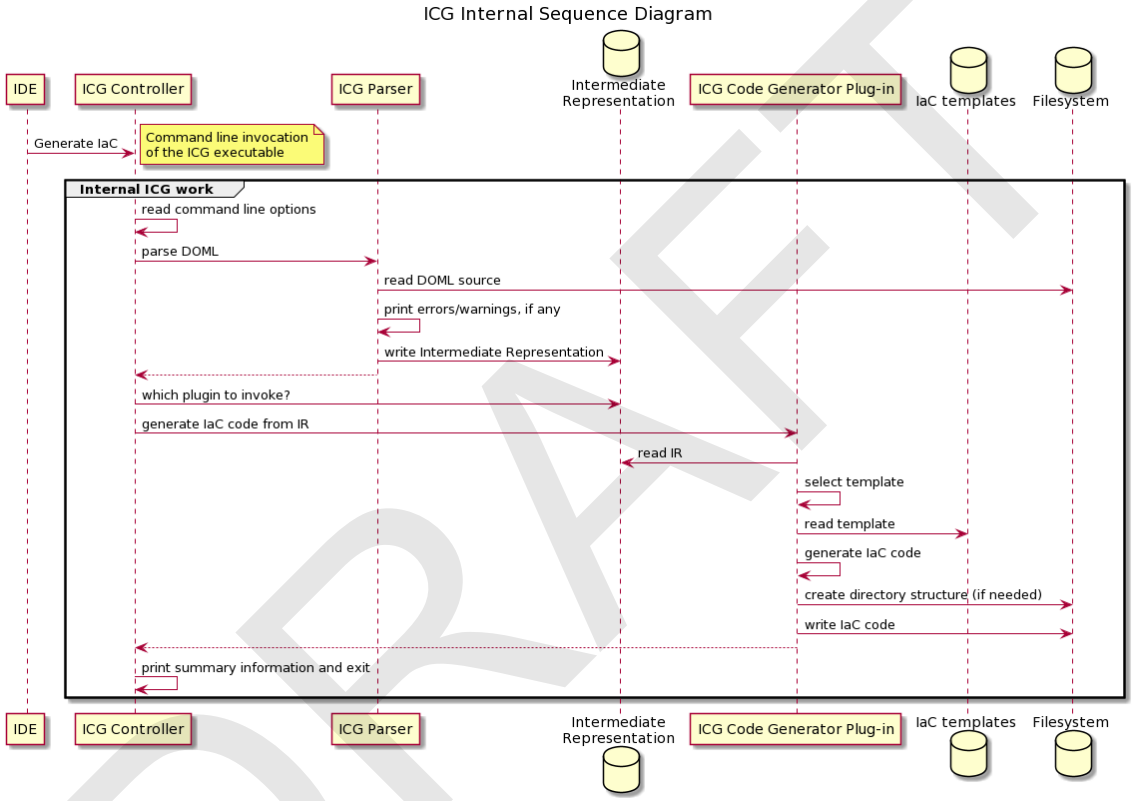


*Figure 8: ICG Internal sequence diagram*

The first component of the ICG is the ICG Controller.

The **Controller** is a Python module dedicated to the management of the processes of the ICG, it is invocated from the command line and receives in input the model generated by the IDE.

The Controller then should call the second component of the ICG, the **Parser**, which takes as an input the DOML model and writes the intermediate representation that the ICG is going to use in the next steps. In this first prototype the parser is not released; we just made an experiment to check if Acceleo would be suitable to implement the Parser, as reported in section 2.1.2.

The last component to be called by the ICG Controller is the **Code Generator**, which is also a module implemented in python. It reads the intermediate representation and, using the IaC templates available for the required resource, it generates the output code. The Code Generator has multiple plug-ins, one for each IaC language that the ICG implements, namely the Ansible plug-in and the Terraform plug-in.

Once the Code Generator plug-in has completed its task, the Controller writes on the console the results of the code generation process and exits.

## 2.2.2 Components' description

### *2.2.2.1 ICG Controller*

The ICG Controller is the core component of the ICG and it manages all other components.

As already specified the Controller is a Python module, integrated with all the other ICG components. In this first implementation it gets the data from the intermediate representation, it analyses the data and based on the contents it calls the correct Code Generator plug-in.

Once the ICG DOML Parser will be complete, the interaction between it and the Controller could be better defined.

```
from ansibleBuilder import *
from doml2terraform import *
import json
import sys

arg_len = len(sys.argv)
if arg_len > 1:
    file_name = sys.argv[1]
else:
    print("Add parameters file name")
    sys.exit()

input_file = open(file_name, "r")
parameters = json.load(input_file)
```

*Figure 9: ICG Controller code piece*

In the Figure 9 above we can see part of the initial implementation of the Controller, which imports the Ansible and Terraform plugins and reads the intermediate representation file which contains the parameters.

### *2.2.2.2 ICG DOML Parser*

The ICG DOML Parser is the component that reads the DOML model and, based on the model, generates the intermediate representation that contains the data used by the ICG to generate the IaC files. The Parser implementation is not yet available in this release; in section 2.1.2 we describe the first experiment done to evaluate a possible implementation.

### *2.2.2.3 Intermediate Representation*

The intermediate representation (IR) is an internal transcription of the essential information from the DOML model, used by the ICG to obtain all the data needed for generating IaC code.

In Figure 10 there is an example of the current version of the intermediate representation.

```
{
    "steps" : [
    {
        "programming_language": "terraform",
        "type":"virtual-machine",
        "output_path": "Output-code/",
        "data":{
            "provider": "aws",
            "network": [
                {
                    "id": 1,
                    "subnetname": "piacere_subnet",
                    "subnet_cidrblock": "10.0.1.0/24",
                    "vpcname": "piacere_vpc",
                    "vpc_cidr": "10.0.0.0/16"
                }
            ],
            "vm": [
                {
                    "id": 2,
                    "os": "ubuntu",
                    "ram": 1,
                    "cpu": 1,
                    "name": "secondvm"
                }
            ],
            "db": [
                {
                    "id": 1,
                    "name": "Danilo",
                    "group_name": "Molteni",
                    "server_name": "DaniloMolteni",
                    "email": "molteni@hpe.com"
                }
            ]
        }
    },{
        "programming_language": "ansible",
        "type":"database",
        "output_path": "Output-code/",
        "info" : {
            "name" : "postgres",
            "template_path" : "Databases-templates/"
        },
        "data":{
            "vars":{
                "DB_USER": "appluser",
                "DB_PASSWORD": "appluser",
                "DB_NAME": "app1"
            },
            "play":{
                "OS": "centos"
            }
        }
    },{
        "programming_language": "ansible",
        "type":"docker-service",
        "output_path": "Output-code/",
        "info" : {
            "name" : "wordpress",
            "template_path" : "Docker-services-templates/"
        },
        "data":{
            "vars": {
                "WORDPRESS_DB_HOST": "10.10.10.10",
                "WORDPRESS_DB_USER": "appluser",
                "WORDPRESS_DB_PASSWORD": "appluser",
                "WORDPRESS_DB_NAME": "app1",
                "WORDPRESS_TABLE_PREFIX": "wp"
            },
            "play":{
                "OS": "centos"
            }
        }
    }]
}
```


*Figure 10: Intermediate representation example*

This version of the intermediate representation used in this example was based on the requirements of the developed IaC templates, and as such it lacks some of the DOML elements (e. g. the interaction between the various components). Future implementations of the Intermediate Representation will enhance the coverage of DOML elements and are expected to be automatically generated by the Parser.

In this version of the IR we have defined a few useful characteristics:

- The various blocks to be generated are listed as a sequence of steps
- In each functional block there are general definitions useful on the operative side and a data segment containing the parameters needed for that functionality
- An example of general definitions is the target IaC language to be used to provide the functionality, or the path where to output the result files; some of this information can be standardized into a default value
- Examples of data are the cloud provider, the operating system installed on the VMs or the resources required for them

The Intermediate Representation file shown in Figure 10 is a JSON file, whose main keywords can be documented as follows.

- *steps*: array of objects representing the sequence of output code blocks to be generated, a different Code Generator plugin may be invoked for each object
- *programming_language*: indicates the type of Code Generator plugin to be invoked for the current step; supported values for this release are “terraform” and “ansible”; if a new plugin is added to generate code for a different output IaC language, the name of the new plugin will be a new possible value for this keyword
- *type*: indicates the type of DOML object for which to generate code; when DOML will be finalized, the possible values for this keyword will be finalized as well
- *output_path*: indicates the path to the directory where the code generated for this object should be stored

- *provider*: indicates which of the supported providers should be used to provision the indicated infrastructure; supported values in this first release are: “aws”, “azure”, and “gcp”; the template selected for code generation depends both on this value and the type of object to be created
- *data*: this keyword contains objects that group the information to be substituted in the selected template’s placeholders, in the form of key/value pairs; the key, e.g. “vpcname”, corresponds to the name of the placeholder, whereas the value, e.g. “piacere_vpc”, corresponds to the value to be substituted when generating code using that template

The syntax of the Intermediate Representation will be homogenized and improved in future releases, to possibly add a stricter relationship to the input DOML model, to simplify the creation of new code generation plug-ins and to support the planned extension functionalities. In future releases the Intermediate Representation will be generated automatically by the Parser.

#### *2.2.2.4 ICG Code Generator*

The ICG Code Generator is the component of the ICG that given the data contained in the model integrates it with the IaC templates available for the required functionalities.

This component is built in Python 3.6 and has multiple separate modules to integrate the multiple plug-ins it provides, one for each IaC language that is integrated in PIACERE. Each plug-in reads the Intermediate Representation (IR) as input and generates the corresponding output code in its respective IaC language. In this release each plug-in is simply called by the Controller with the IR as an input parameter; in future releases, we plan to define a more complete and flexible common plug-in interface, to facilitate the integration of new plug-ins targeting the support of new IaC languages.

An essential element for the code generation is the set of IaC templates, which provide the vocabulary for the transcription from the DOML to the destination IaC language. Each template is composed by static text in the target IaC language and placeholders, which are substituted with actual values at code generation time. The values are either taken from the input Intermediate Representation or generated dynamically based on defaults.

For the scope of this prototype, we have an initial Terraform plug-in prototype and an Ansible plug-in prototype. The selection of the right plug-in to use is done by the ICG Controller, based on the information included in the Intermediate Representation. In this first release, we decided to generate different languages, depending on the activity to be automated: Terraform code for provisioning activities and Ansible code for installation / configuration activities.

##### 2.2.2.4.1 Terraform plug-in

The Terraform plug-in is tasked with the generation of the Terraform IaC files.

In this first iteration of the prototype, the Terraform code is used for the deployment functionalities of PIACERE. It manages provisioning for both network and virtual machines on the selected cloud platforms (at the moment AWS, Azure and Google Cloud Platform are supported).

In Figure 11 there is a representation of a Terraform code template, as was previously described this is the basis for the transcription from DOML to Terraform.

```
data "{{ vm }}" "ami{{ id }}" {
  #executable_users = {{ executable_users }}
  most_recent = {{ mostrecent }}
  name_regex = "{{ name_regex }}"
  #owners = {{ owners }}
  {{ filters }}
  owners = ["099720109477"] # Canonical
}
resource "aws_instance" "instance{{ id }}" {
  ami = data.aws_ami.ami{{ id }}.id
  instance_type = "{{ instance_type }}"
    tags = {
     Name = "{{ name }}"
    }
}
```

*Figure 11: Terraform template example*

In this template the delimiters used to define the input parameters are the classic jinja2 delimiters, so these templates are readable and can be compiled by any jinja2 compatible software, provided the input variables.

An example of input parameters are the `{{ id }}` or the `{{ instance_type }}`.

There are different templates for the different cloud providers and there are separated templates for the various functionalities (e.g. Network deployment, VM deployment) a more complete list of the currently available templates can be found in section 3.1.

#### 2.2.2.4.2 Ansible plug-in

The Ansible plug-in is tasked with the generation of the Ansible IaC files.

In this first iteration of the prototype, the Ansible code is used for the configuration functionalities of PIACERE. It manages the code that configures the services and applications on the provided virtual machines (for this first prototype was implemented code for the configuration of applications running on Docker, namely WordPress, and databases, namely Postgres).

In Figure 12, there is a representation of an Ansible code template.

```
- hosts: APP1
  become: yes

  vars_files:
    - wordpress-vars.yml

  pre_tasks:
    - name: "Install packages"
      ###OS###: "name={{ item }} state=present"
      with_items:
###OS_PACKETS###      Niculut Laurentiu, a week ago

    - name: "Install Python packages"
      pip: "name={{ item }}  state=present"
      with_items:
        - docker

  tasks:
    - name: Start a WP container
      community.docker.docker_container:
        name: wordpress
        image: wordpress:5.8.0
        state: started
        env:
          WORDPRESS_DB_HOST: "{{WORDPRESS_DB_HOST}}"
```

*Figure 12: Ansible template example*

In this example template we have 2 kinds of parameters, the first kind are the parameters that the playbook gets from the vars.yml file (which is also built by the ICG) and the second kind are the parameters that enable the template to operate on different environments. The difference between these 2 types of parameters is that the first ones make use of the functionalities of Ansible in an optimal way, while the second ones allow to expand the standard capabilities of Ansible adding functionalities that were not present before.

Here is an example of the 2 types of parameters available:

- First: `{{WORDPRESS_DB_HOST}}`
- Second: `###OS_PACKETS###`

There are different templates for the different functionalities that are provided through Ansible; as already stated for the Terraform plug-in, a more complete list of the currently available templates can be found in section 3.1.

### 2.2.3 Technical specifications

In this section the technical specifications of the current ICG prototype are defined. Please note that these specifications are likely to change in future releases.

The ICG Controller and Code Generator are both written in Python 3.6. As such the main prerequisites for this prototype are:

- Python version 3.6
- Jinja2 Python library version 3.0.3

As for the output IaC code it is generated for:

- Terraform version 1.0.10
- Ansible version 4.6.0

# 3 Delivery and usage

## 3.1 Package information

The structure of the files and folders of the ICG can be seen in the following Figure 13.

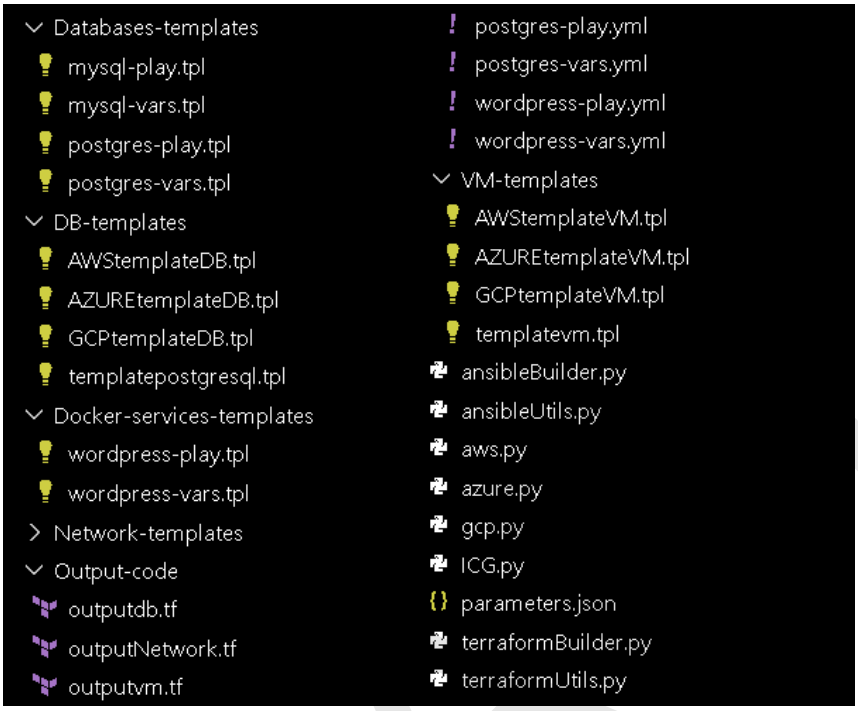


*Figure 13: ICG folder and file structure*

The folders are of two kinds, one folder contains the IaC code generated by the ICG:

- Output-code

The output directory can be configured, and its path is defined by the keyword “output_path” in the intermediate representation.

The other folders contain the templates used to generate the IaC output files:

- Databases-templates: contains the Ansible templates to install and configure supported databases (Postgres, MySQL) on existing Virtual Machines
- DB-templates: contains the Terraform templates to provision DB as a service on the supported cloud providers
- Docker-services-templates: contains the Ansible templates to deploy services on Docker containers
- Network-templates: contains the Terraform templates to provision network infrastructure on different cloud providers
- VM-templates: contains the Terraform templates to provision virtual machines on the supported cloud providers

Each of these folders contains multiple files. The **Output-code** folder contains all the Terraform and Ansible code files that are generated by the ICG, in the Figure 13 above we can see the files generated for the Wordpress example:

- outputdb.tf
- outputNetwork.tf

- outputvm.tf
- postgres-play.yml
- postgres-vars.yml
- wordpress-play.yml
- wordpress-vars.yml

All the other folders contain the templates used by the ICG. The templates available in this first prototype are documented below, folder by folder.

Folder 1) The first folder is **Database-templates** and it contains four files:

- mysql-play.tpl
- mysql-vars.tpl
- postgres-play.tpl
- postgres-vars.tpl

These four templates are used to generate the Ansible code necessary to deploy a Postgres or MySQL database on a virtual machine. The vars templates take as an input the parameters provided through the intermediate representation and generate the vars file useful to the Ansible execution. The play template is instead the main code of the Ansible playbook containing the Ansible pre-tasks and tasks.

These playbook templates, aside from the standard parameters provided through the vars file, also have supplemental parameters that allow the ICG to generate Ansible code that can be executed both on CentOS and on Debian operating systems.

Folder 2) The second folder is **DB-templates** and it contains four files:

- AWStemplateDB.tpl
- AZUREtemplateDB.tpl
- GCPtemplateDB.tpl
- templatepostgresql.tpl

These are the templates implemented to generate Terraform code that deploys database services on the cloud providers, the templates are one for each cloud provider implemented as of this version of the ICG (AWS, Azure and Google Cloud Platform).

In the Figure 14 below, we can see one of these templates, the AWS one.

```
resource "aws_db_instance" "{{ identifier }}" {
  identifier             = "{{ identifier }}"
  instance_class         = "{{ instance }}"
  allocated_storage      = {{ storage }}
  engine                 = "{{ engine }}"
  engine_version         = "{{ version }}"
  username               = "{{ username }}"
  password               = {{ password }}
  db_subnet_group_name   = {{ subnet }}
  vpc_security_group_ids = {{ security }}
  parameter_group_name   = {{ parameter }}
  publicly_accessible    = {{ accessible }}
  skip_final_snapshot    = {{ skip }}
}
```

*Figure 14: Current implementation of AWStemplateDB.txt*

Folder 3) The folder **Docker-services-templates** contains two files as of this implementation:

- wordpress-play.tpl
- wordpress-vars.tpl

These two templates are used to generate the Ansible code necessary to deploy a WordPress web application on a virtual machine. The vars template and play template have the same functionalities as described for the MySQL and Postgres ones. They also have the supplemental parameters that allow ICG to generate Ansible code that can be executed both on CentOS and Debian.

The deployment strategy that was selected for this implementation is to have all applications running on Docker, so this playbook deploys the Docker service on the VM and then deploys the WordPress container; this solution can be easily replicated for other applications.

Folder 4) The folder **Network-templates** contains three files:

- AWStemplateNetwork.tpl
- AZUREtemplateNetwork.tpl
- GCPtemplateNetwork.tpl

These are the templates implemented to generate Terraform code that deploys the network infrastructure on the cloud providers, the templates are one for each cloud provider supported as of this version of the ICG (AWS, Azure and Google Cloud Platform).

In the Figure 15 below, we can see one of these templates, the GCP one.

```
resource "google_compute_network" "vpc_network" {
    name = "{{ network }}"
}
resource "google_compute_subnetwork" "public-subnetwork" {
    name = "{{ subnetwork }}"
    ip_cidr_range = "10.2.0.0/16"
    region = "us-central1"
    network = google_compute_network.vpc_network.name
}
```

*Figure 15: Current implementation of GCPtemplateNetwork.txt*

Folder 5) The folder **VM-templates** contains four files:

- AWStemplateVM.tpl
- AZUREtemplateVM.tpl
- GCPtemplateVM.tpl
- templatevm.tpl

These are the templates implemented to generate Terraform code that deploys the virtual machines on the cloud providers, the templates are one for each cloud provider implemented as of this version of the ICG (AWS, Azure and Google Cloud Platform).

In the Figure 16 below, we can see one of these templates, the Azure one.

Aside from this folder there are also seven separate files that are available in the package, one of them is the parameters' file and the remaining six contain the ICG code written in Python.

The parameters' files is the following:

- parameters.json

```
terraform {
  required_providers {
    azurerm = {
      source = "{{ source }}"
      version = "{{ version }}"
    }
  }
}
provider "azurerm" {
  features {}
}
resource "azurerm_resource_group" "rg" {
  name = "{{ name }}"
  location = "{{ location }}"
}
```

*Figure 16: Current implementation of AZUREtemplateVM.txt*

The `parameters.json` file contains the intermediate representation (see section 2.2.2.3 for more details) that is used as an input for this prototype of the ICG. Due to the bottom-up approach of the development of this prototype, this file is at the moment edited manually and it needs further development to better represent the DOML model. In future releases this file will be generated automatically by the Parser.

At last, the **ICG source code** is composed of the following files:

- aws.py
- azure.py
- gcp.py
- ansibleBuilder.py
- ansibleUtils.py
- terraformBuilder.py
- terraformUtils.py
- ICG.py

The `ICG.py` file is the main module of the ICG and contains all the functionalities provided by the ICG Controller.

Then there are the two plug-ins, one for Terraform and the other for Ansible.

The main code of the Ansible plug-in can be found in the `ansibleBuilder.py` file, while the main module of the Terraform plug-in can be found in the `terraformBuilder.py` file.

Aside from the main module, each of the plug-ins has additional files with helper functions. For the Ansible plug-in, the file `ansibleUtils.py` contains all the necessary functions. Whereas for the Terraform plug-in, the file is `terraformUtils.py`. Finally, other source files, `aws.py`, `azure.py` and `gcp.py`, provide specific functions needed for the respective provider.

## 3.2 Installation instructions

The prototype provided is constructed on Python 3.6 so the environment on which it will run shall have Python 3.6 installed.

The Python library Jinja2 is needed to run the ICG, so it shall be installed trough pip with the following command:

```
pip install Jinja2==3.0.3
```

Inside the python 3.6 environment can then be downloaded the prototype package which contains all the folders and files needed to run the example. The package, provided the right permissions on the GitLab repository, can be downloaded trough the following command:

```
git clone –b develop https://git.code.tecnalia.com/piacere/private/icg-infrastucture-code-generator/t34-icg-controller.git
```

Once the package is downloaded, the prototype is ready to be executed.

## 3.3 User Manual

The initial step of configuring the environment and installing the package is fully detailed in the chapter 3.1, once it is all installed the prototype ca be tested.

The ICG is a command line executable as such it is invocated either from a shell in linux or a windows terminal.

The command to be executed from the package main folder is the following:

```
.\ICG.py parameters.json
```

The file `ICG.py`, as already specified, is the main module of the ICG and the `parameters.json` file contains the (hand generated) intermediate representation of the model. In the next release the intermediate representation will be generated by the DOML Parser and will be automatically used by the Controller, without the need for specifying it on the command line.

With this invocation the ICG is going to generate the IaC code as specified in the `parameters.json` file, for this prototype to get a different output code (for a different provider, operating system, or other specifications) the parameters file has to be modified manually.

If the execution runs correctly no output will be visible on prompt, otherwise error messages will be printed on screen.

Once the code is executed correctly the output code files will be generated by the ICG and can be found in the Output-code folder noted in section 3.1.

## 3.4 Licensing information

The plan is to release the ICG component, developed by HPE, as open-source software. For this, HPE must follow an internal process with reviews and decisions at corporate level to decide and approve the license under which to release the developed software. Unfortunately, this process takes a lot of time and it is not yet even started at the time of writing, therefore the licensing information for the released software is still to be defined.

## 3.5 Download

The Infrastructural Code Generation code is available in the PIACERE code repository at: https://git.code.tecnalia.com/piacere/private/icg-infrastucture-code-generator/t34-icg-controller.

The source code will be available on the public git repository and accessible through the project's website https://www.piacere-project.eu/. At the time of writing this deliverable, the source code is provided under request through an email to the address appearing on the website (https://www.piacere-project.eu/) in the footer under "Contact Us".

# 4 Conclusions

This document described the first ICG prototype that has been implemented in the first year of the PIACERE project. As already explained in section 1.1, the adopted bottom-up approach led to a prototype that is more focused on the code generation functionalities than rather on parsing the input DOML language.

The main functionalities of the ICG have been listed, along with its internal architecture, the relationship among the planned functionalities and the requirements collected in deliverable D2.1, and the experiment done to evaluate a possible implementation of the Parser component.

All internal ICG components have been described, along with the prerequisites for their execution. Furthermore, the deliverable describes in detail the files included in the delivered package and documents how to install and use the released software.

Future versions of the ICG will complete the planned functionalities and will improve several aspects of the component. In the next version, due at M24 (deliverable D3.5), the DOML Parser component will be developed, in order to read DOML and automatically generate the Intermediate Representation. The syntax of the Intermediate Representation itself will be homogenized and improved, to possibly add a stricter relationship to the input DOML model, to enhance the coverage of DOML elements, to simplify the creation of new code generation plug-ins and to support the planned extension functionalities. In the final release, due at M30 (deliverable D3.6), implementing ICG extensibility, we will also provide guidelines for writing new templates, so that expert users will be able to develop their own templates or to modify existing ones, both for supporting new DOML concepts and for providing support for new IaC languages. The plan for extensibility is also to define a more complete and flexible common plug-in interface, in order to facilitate the integration of new plug-ins targeting the support of new IaC languages. Finally, more templates will be defined, already in the next version at M24, both for improving the support of DOML elements and possibly for supporting other target platforms, depending on Use Case requirements.

# 5 References

[1] Syriani, E., Luhunu, L., & Sahraoui, H. (2018). *Systematic mapping study of template-based code generation*. Computer Languages, Systems & Structures, 52, 43-62.