PIACERE

Deliverable D3.6

Infrastructural code generation – v3

Editor(s):	Debora Benedetto, Lorenzo Blasi, Laurentiu Niculut
Responsible Partner:	HPE, HPECDS
Status-Version:	Final - v1.0
Date:	29.05.2023
Distribution level (CO, PU):	PU

Project Number:	101000162
Project Title:	PIACERE

Title of Deliverable:	Infrastructural code generation – v3
Due Date of Delivery to the EC	31.05.2023

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

Abstract:	This deliverable presents the advancements of Task T3.4 made from M24 to M30 and is the final deliverable version for ICG. It comprises both an updated version of the software prototype [KR3] and a Technical Specification Report. The document has a new structure, common to all deliverables documenting software released at M30, and includes in the Appendix the technical design of the current version of the ICG, installation instructions and user manual.
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

Document Description

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
v0.1	31.03.2023	Update ToC	HPECDS
v0.2	16.05.2023	Document fully reviewed, ready for internal review and candidate to the final version v1.0	HPE, HPECDS
v0.3	22.05.2023	Updates after the internal review	HPE
v0.4	23.05.2023	Updated to be more self-consistent	HPE
v0.5	29.05.2023	Updated with the new ICG sequence diagram	HPE
v0.6	29.05.2023	Final version ready for release	HPE
v1.0	31.05.2023	Final quality check. Ready for submission	TECNALIA

Table of contents

Terms and abbreviations 6
Executive Summary 7
1 Introduction 8
1.1 About this deliverable 8
1.2 Document structure 8
2 KR 3-ICG overview 9
2.1 Changes in v3 9
2.2 Functional description and requirements coverage 9
2.3 Main innovations 11
3 Overview of preliminary experiments 12
3.1 UC1: Slovenian Ministry of Public Administration 12
3.2 UC2: Critical Maritime Infrastructures 12
3.3 UC3: Public Safety on IoT in 5G 13
4 Lessons learnt and outlook to the future 14
5 Conclusions 16
6 References 17
APPENDIX: Implementation, delivery and usage 18
1 Implementation 18
1.1 Fitting into overall PIACERE Architecture 18
1.2 Technical description 19
1.2.1 Prototype architecture 19
1.2.2 Components description 20
1.2.3 Output description 23
1.2.4 Technical specifications 24
2 Delivery and usage 25
2.1 Package information 25
2.2 Installation instructions 25
2.3 User Manual 26
2.3.1 ICG usage from REST API 26
2.3.2 ICG usage from command line 27
2.3.3 ICG extension: writing a new template 27
2.3.4 ICG extension: adding a new target language 29
2.4 Licensing information 31
2.5 Download 31

List of tables

TABLE 1: KR3 - ICG REQUIREMENTS 10

List of figures

FIGURE 1: VMWARE VSPHERE TEMPLATES 12
FIGURE 2: ICG SEQUENCE DIAGRAM 18
FIGURE 3: ICG INTERNAL ARCHITECTURE 19
FIGURE 4: ICG INTERNAL SEQUENCE DIAGRAM 20
FIGURE 5: ICG PARSER - NEW IAC LANGUAGE 21
FIGURE 6: ICG PARSER - NEW STEP FOR NEW IAC LANGUAGE 22
FIGURE 7: NEW CROSS-PLATFORM FOLDER 23
FIGURE 8: ICG OUTPUT 23
FIGURE 9: ICG REST API 26
FIGURE 10: ICG POST /IAC/FILES/UPLOAD EXAMPLE 27
FIGURE 11: ICG POST /IAC/FILES/UPLOAD EXAMPLE OUTPUT 27
FIGURE 12: NEW GENERATED TEMPLATE 29
FIGURE 13: NEW TEMPLATES IN ICG PROPERTIES FILE 29
FIGURE 14: ICG EXTENSION - NEW DOCKER-COMPOSE SECTION 30
FIGURE 15: ICG EXTENSION – INTERMEDIATE REPRESENTATION WITH DOCKER-COMPOSE RESOURCES 31
FIGURE16: ICG EXTENSION – DOCKER-COMPOSE PLUG IN 31

Terms and abbreviations

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
IaC	Infrastructure as Code
IcaaS	IaC code as a Service
ICG	Infrastructural Code Generator
IEM	IaC Executor Manager
IOP	IaC Optimizer Platform
IR	Intermediate Representation
KPI	Key Performance Indicator
KR3	Key Result 3
PRC	PIACERE Runtime Controller
UC	Use Case
SW	Software

Executive Summary

This deliverable describes the third and last version of the PIACERE Infrastructural Code Generator (ICG), developed in Task 3.4. The ICG is one of the main innovations resulting from the PIACERE project. While the DOML (DevOps Modelling Language) model describes the desired infrastructure and its characteristics, the ICG generates IaC code that realizes both provisioning and configuration of the target infrastructure and enables "bringing it to life" in the exact way that has been specified in the model.

The document has a new structure, common to all deliverables documenting software released at M30. In Section 3, it provides a comprehensive overview of the ICG (KR3), reports about improvements developed in the last half year of the relevant task, updates about requirements coverage and describes the main ICG innovations. Section 4 provides some lessons learnt, a view of possible future enhancements and next steps for further research on IaC generation.

The document also contains, in the Appendix, a description of the ICG implementation and how it fits into the overall PIACERE framework, plus final documentation about installation, usage and extension of the ICG.

This release of ICG improves over the second version by integrating with the PIACERE security agents, by extending the template library to support new cloud providers and a new orchestrator; the ICG API has also been updated to give the user the possibility to include their own IaC files in the generated output. Finally, this ICG version also produces a Gaia-X "self-description" for the generated IaC code, to allow loading it into a Gaia-X repository, along with the related DOML source.

ICG can be extended in multiple directions. The current version generates both Terraform, Ansible and Docker Compose IaC code, but is extensible to generate new target infrastructure languages, as documented in section 2.3.4. ICG currently generates Ansible code for both CentOS and Ubuntu but is easily extensible to generated code for other target operating systems, by adding new templates as explained in section 2.3.3. The generated Terraform code already supports several virtual infrastructure providers (AWS, Azure, Google Cloud, IONOS, OpenStack, vSphere), but support for other providers can be added.

In conclusion, ICG is a highly extensible and flexible tool that can be useful in multiple environments and situations.

1 Introduction

1.1 About this deliverable

This deliverable is the third and last version of the deliverable about the Infrastructural Code Generator (ICG) component of PIACERE. The document provides a comprehensive overview of the ICG (KR3), reporting about updates developed in the last half year of the relevant task, a functional description updating about requirements coverage, some lessons learnt and a view of possible future enhancements. The Appendix contains a description of the ICG implementation and how it fits into the overall PIACERE framework, plus documentation about installation, usage and extension. This release of ICG improves over the second version (D3.5 [1]) by integrating with the PIACERE security agents, by extending the template library to support new cloud providers (IONOS [2] and VMWare [3]) and a new orchestrator (Docker Compose [4]); the ICG API has also been updated to give the user the possibility to include their own IaC files in the generated output. Finally, this ICG version also produces a Gaia-X “self-description” for the generated IaC code, to allow loading it into a Gaia-X repository, along with the related DOML source.

1.2 Document structure

The document has a new structure, common to all deliverables documenting software released at M30. After this Introduction, section 2 provides an overview of the released ICG component (KR3), reporting about changes in this release, a functional description updating about requirements coverage, and the main ICG innovations; section 3 explains the main experiments performed and the challenges encountered; section 4 highlights the lessons learnt and future work for this KR, and section 5 draws the conclusions. The APPENDIX contains information about implementation, delivery and usage, focusing on changes in this release. Section 1 of the APPENDIX describes how the ICG prototype fits into the overall PIACERE Architecture and how it is related with other components, reports about the ICG prototype implementation, with a description of its architecture and of internal ICG components. D3.6 is a software deliverable, therefore this document also provides details about the released software in section 2 of the APPENDIX: how it is structured, how it can be obtained, installed and used, plus licensing information.

2 KR 3-ICG overview

The Infrastructural Code Generator (ICG) is the PIACERE tool responsible for the Infrastructure as Code (IaC) generation.

In PIACERE, the user defines their application using the abstract DOML language. The ICG then translates this definition choosing the proper IaC languages, creates the files and organizes them adding metadata to let the execution work properly. Moreover, the ICG supports multiple cloud platforms. In this way, the user can write their application in DOML for one or more providers in a more abstract definition, and the ICG is going to take care about the proper translation and language to be used.

Thus, the ICG makes the IaC definition easier for the user, who does not have to take care to choose the correct IaC language or learn each of them. Furthermore, the user can deploy the same application on different platforms and change the provider in an easier way, without going deeper into the providers details and learn each specific resource available.

2.1 Changes in v3

The third version of the ICG has been released at M30 and it contains some improvements and refinements with respect to the previous version in M24 (D3.5 [1]).

The ICG runs both as command line tool and as Docker container. It is called from the PIACERE IDE and produces as output a zip folder containing the IaC files and instruction on how to execute them. The ICG API has been updated to extend the PIACERE framework functionalities and gives the user the possibility to add their own IaC files in the generated IaC structure. This feature has been added to enable reuse of existing code as requested by the new requirement REQ111. In the new API the ICG now takes as input a zip folder containing the DOML model and, optionally, extra user's IaC files. It generates (from the DOML model) the corresponding IaC files and adds them to the user's extra code. Finally, the ICG packs everything according to the PIACERE format and sends back to the IDE the new zipped folder.

The integration with the PIACERE security agents is now available and the ICG output provides IaC files to install the agents on every Virtual Machine defined in the user's infrastructure.

The template library was extended to support new providers: IONOS [2] and VMWare [3] cloud providers, and a new orchestrator: Docker Compose [4]. Configuration IaC code for new applications, such as Elasticearch[1], can now be generated thanks to the definition of new Ansible templates.

Finally, the ICG predisposed its output to make it possible the integration with the Gaia-X [5] European initiative. Thus, the ICG outcomes contains a "self-description" file and the IONOS [2] IaC files supported by this project.

2.2 Functional description and requirements coverage

The ICG architecture has not been changed and its full description is presented in this deliverable in Section 1.2.1.

The ICG main functionalities are not changed respect to the D3.5 [1] and we recap them below:

- F1. Read the input DOML model to extract all the needed information.
- F2. Generate executable code for selected IaC languages.

[1] https://www.elastic.co/

- 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.

In this third release, the functionalities F1, F2 and F5 have been extended and the others are implemented.

The following table reports the updates about the coverage of each ICG requirements indicated in deliverable D2.2 [6] in relation to the functionalities.

Requirements in Table 1 have been updated and fulfilled: REQ96 is achieved thanks to the update of the ICG DOML Parser and template library, REQ77 is achieved and supports the deployment step thanks to the Docker Compose orchestrator, REQ31 was achieved and now extended with the new Docker Compose IaC language, REQ100 was achieved and now extended supporting IONOS and VMWare cloud platforms, REQ110 is now achieved and supports the DOML and ICG extension mechanism, REQ111 is a new requirement introduced to support the integration of a user's external folder and it is successfully achieved.

Table 1: KR3 - ICG Requirements

Funct.	Req ID	Description	Status	Requirement Coverage at M24
F1	REQ96	ICG must be able to read DOML language.	Achieved	ICG can parse the DOML v3
F2	REQ31	ICG should provide verifiable and executable IaC generated from DOML for selected IaC languages (e.g., TOSCA/Ansible/Terraform).	Achieved	Already implemented in v1, see previous deliverable (D3.4 [7]). Extended with the Docker Compose IaC language.
F2	REQ77	ICG may generate IaC code for different supported/target tools according to the required DevOps activity (as listed in REQ76).	Achieved	Orchestration has been implemented using Docker compose.
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.	Discarded	See previous deliverable D3.5 [1].
F3	REQ110	ICG should provide enough extensibility to: comply with the DOML extension mechanism; be capable of integrating new IaC languages	Achieved	The ICG supports the DOMLv3.0 extension mechanism.
F4	REQ110	ICG should provide enough extensibility to: comply with the DOML extension mechanism; be capable of integrating new IaC languages	Achieved	The ICG implements extension mechanism to integrate new IaC languages.
F4	REQ111	The user could have the possibility to add external custom own IaC.	Achieved	ICG implemented a new API compatible with the addition of external custom IaC.
F5	REQ29	DOML should support the modelling of VM provisioning for different platforms such as	Discarded	See previous deliverable D3.5 [1].

		(OpenStack, AWS) for canary and production environments.		
F5	REQ100	ICG should generate IaC code that supports different cloud platforms.	Achieved	See previous deliverable D3.5 [1]. Now ICG supports IONOS and VMWare cloud platforms too.

2.3 Main innovations

The Infrastructural Code Generator (ICG) is one of the main innovations resulting from the PIACERE project. When the DOML (DevOps Modelling Language) model describes the desired infrastructure and its characteristics, the ICG generates IaC code that realizes both provisioning and configuration of the target infrastructure and enables "bringing it to life" in the exact way that has been specified in the model.

Along with IaC code for provisioning the infrastructure, ICG also generates IaC files to install both the PIACERE security agents and PIACERE monitoring agents on every Virtual Machine defined in the target infrastructure, thus making it observable and more secure. In addition ICG can also include in the output multiple IaC files provided by the user, for additional flexibility and to enable reuse of existing code.

Further flexibility is provided by the choice of multiple usage modes. ICG can be installed locally by every developer and used from the command line as explained in section 2.3.2, but it can also be installed as a shared server and used through its REST API. This opens the way to providing the generation of IaC code as a Service (IcaaS).

The current version of ICG generates Terraform, Ansible and Docker compose IaC code, but is extensible to generate new target infrastructure languages, as documented in section 2.3.4. ICG currently generates Ansible code for both CentOS and Ubuntu but is easily extensible to generated code for other target operating systems, by adding new templates as explained in section 2.3.3. The generated Terraform code already supports several virtual infrastructure providers (AWS, Azure, Google Cloud, IONOS, OpenStack, vSphere), but support for other providers can be added.

The innovations described in this section make ICG a highly extensible and flexible tool that can be useful in multiple environments and situations.

3 Overview of preliminary experiments

The main experiments done using the ICG are centred around the Use Cases implementation. The use cases provided were the following three and each of them brought useful insights:

- UC1: Slovenian Ministry of Public Administration
- UC2: Critical Maritime Infrastructures
- UC3: Public Safety on IoT in 5G

Further details on all these use cases are presented in the next paragraphs.

3.1 UC1: Slovenian Ministry of Public Administration

The "Slovenian Ministry of Public Administration" use case (UC1) is the largest use case between the three available, with multiple applications connected.

First of all, this use case has a strict requirement regarding the provider to be used, indeed it only supports VMware vSphere and for this reason this provider was added into all the PIACERE framework: the Infrastructural Elements Catalogue, the DOML, the IEM and the ICG are the components involved. The ICG had to add multiple templates to support the complete functionality of this provider, comprising network functionalities, storage and computing resources.

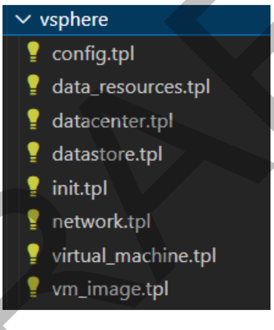

```
vsphere
  config.tpl
  data_resources.tpl
  datacenter.tpl
  datastore.tpl
  init.tpl
  network.tpl
  virtual_machine.tpl
  vm_image.tpl
```

Figure 1: VMware vSphere templates

The addition of the new provider was done taking advantage from the extension mechanism detailed in the user manual appendix in Section 2.3.

This use case also implements multiple proprietary applications and thus it was necessary to add a new requirement (REQ111). This new requirement is related to the possibility for the user to add custom IaC files for all those applications that couldn't easily be modelled with the available PIACERE resources. In particular, it states that the user should be capable of adding a folder inside the PIACERE IDE workspace containing the custom IaC code to be added. These external files should be then security checked by the IaC Scan Runner and executed by the IEM. This requirement states that the ICG should add the external IaC files to the generated code in the output folder.

To achieve this, multiple updates have been done, starting with a new API capable of receiving the external code (Section 2.3.1), the logic to add this custom code to the generated one and the new updated output (Section 1.2.3).

3.2 UC2: Critical Maritime Infrastructures

The "Critical Maritime Infrastructures" use case (UC2) was the first one to be used to experiment on the ICG. Since the complexity required is less than the other two, it provided very useful insights during the initial process of integration with the other PIACERE components.

This use case runs on the AWS provider, it uses Amazon Elastic Compute Cloud instances to run multiple applications connected to each another, it also uses the autoscaling groups provided by AWS to handle infrastructural resource optimization.

This use case in the first place was very useful to expand our Terraform template library for AWS and our Ansible template library with applications as ElasticSearch. Between the notable templates addition, there is the autoscaling groups concept, this came with the definition of an entirely new dedicated resource in the DOML model, so also the ICG had to add this new resource with dedicated templates. By doing this, the concept of autoscaling groups can now be easily exported to the other providers that have this kind of resource.

This use case was also extensively used to test the integration with the other components, so it was useful during the definition of all the complementary files necessary for the correct execution of the generated code.

3.3 UC3: Public Safety on IoT in 5G

The “Public Safety on IoT in 5G” use case (UC3) provided some interesting technological challenges that have brought interesting insights.

This use case runs on the AWS provider, an already implemented provider, and uses containerized applications with pre-provided images, so all the infrastructural definitions where almost all already available. The complexity of this use case was in the necessity of orchestrating all the containers that need correct definitions and connectivity with one another.

To handle this necessity a new IaC language was added to the ICG, this process is detailed in the appendix Section 2.3.4. This process also was done in collaboration with the DOML and IEM as the first one had to correctly describe these resources, while the second needed to be able to execute this new language. The ICG from its side has to correctly parse what is provided by the DOML and generate the correct files for the IEM to be able to execute the code.

4 Lessons learnt and outlook to the future

This section summarizes lessons learnt during the whole work done since the beginning of the PIACERE project. We highlight what went well and what didn't, the challenges we encountered, strengths and weaknesses, plus some consideration on what could have been done better. At the end we also indicate some research directions to improve the ICG that could be pursued in the last months or after the end of the project.

One first challenge that we had to deal with at the beginning of the project was the need to create a translator from DOML to IaC code, while DOML was still being defined. We solved this problem by defining a modular architecture for the ICG, with a clear separation from the front-end, the Parser reading the DOML language, and the back-end generating IaC code. An Intermediate Representation was defined as the hinge between the Parser and the Code Generator, to allow postponing the Parser work to a later time, when the DOML definition achieved some stability. This choice allowed us to develop code generators for both Terraform and Ansible already in the first year of the project.

Another challenge was the development of IaC code, or templates, for a specific provider without direct access to its services. This problem was solved thanks to the good collaboration and communication with other PIACERE partners: those with access to a provider helped us by testing the IaC code that we produced through blind programming.

Further challenges, better explained in the paper that we presented to the ICPE’23 conference [8], were “the transformation of complex and interrelated objects and their dependencies into code”, solved by carefully defining templates, the obscuration of sensitive data, solved by storing secrets in the Vault[2] component, and the selection of the right target language for each task, currently solved by defining defaults: Terraform for the infrastructure layer and Ansible for the application layer.

Some possible improvements and enhancements that we think could be indicated as research directions for the future work on the ICG components are the following.

- The ICG Controller could be refactored by defining a standardized API to connect it to the code generation plug-ins, so that the development of new plug-ins could be simplified, up to avoiding the modification of the existing code.
- A plug-in could be developed to integrate ICG into Jenkins[3], so that the generation of IaC code can be invoked in CI/CD pipelines. This could for example provide more automation for groups working together on a single DOML model: each commit operation in git, adding a new version of the common model, could be used as a trigger to start a Jenkins pipeline that verifies the DOML and then invokes ICG to create the corresponding new IaC code.
- An additional parser for the TOSCA language could be integrated in the ICG to produce the same Intermediate Representation, and a DOML representation in TOSCA could be defined as a TOSCA extension. This could allow using TOSCA-based graphical tools, such

[2] https://www.vaultproject.io/
[3] https://www.jenkins.io/

as Eclipse Winery[4], to specify the target infrastructure that will be created by the IaC code generated by the ICG.

- Further target languages can be added, as indicated in section 2.3.4. Adding for example Kubernetes[5] (K8s), would allow the direct deployment of applications as containers in K8s Pods, but could also allow the creation of virtual machines by exploiting the KubeVirt[6] K8s extension. The target K8s cluster itself could be even created in an automated way by using IaC code generated by ICG.

[4] https://projects.eclipse.org/projects/soa.winery
[5] https://kubernetes.io/
[6] https://kubevirt.io/

5 Conclusions

This deliverable describes the status of the ICG component at the end of its development cycle, the component is fully operational and has implemented multiple IaC languages, providers and resources, still there are possible future developments and the component supports multiple types of extensibilities to allow new developments.

With the release of this deliverable the v3.0 of the ICG is being released that is compatible with the last official release of DOML that also ends its development.

The ICG, as of this release, is fully integrated with the PIACERE components that interact with it like the IDE, the IEM and the monitoring and security agents, but it's also integrated with external tools such as Gaia-X.

From the last deliverable, as it was left, guidelines for writing new templates have been provided, to allow the expert user to extend ICG as much as needed.

6 References

[1] PIACERE Consortium, “PIACERE Deliverable D3.5 - Infrastructural code generation – v2,” [Online]. Available: https://zenodo.org/record/7431184#.Y6GM0nbMI2w.

[2] “IONOS Cloud Provider,” [Online]. Available: https://cloud.ionos.com/.

[3] “VMWare,” [Online]. Available: https://www.vmware.com/.

[4] “Docker Compose,” [Online]. Available: https://docs.docker.com/compose/.

[5] “gaia-x,” [Online]. Available: https://gaia-x.eu/.

[6] PIACERE Consortium, “PIACERE Deliverable D2.2 - DevSecOps Framework Requirements specification, architecture and integration strategy - v2,” [Online]. Available: https://zenodo.org/record/7430090#.Y5gjkXbMI2w.

[7] PIACERE Consortium, “PIACERE Deliverable D3.4 - Infrastructural code generation – v1,” [Online]. Available: https://zenodo.org/record/6821657#.Y0_Gr3ZBw2x.

[8] Nedeltcheva G., Xiang B., Niculut L., Benedetto D., “Challenges Towards Modeling and Generating Infrastructure-as-Code,” *In Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (pp. 189-193),* 2023, April.

[9] “Git Submodules,” [Online]. Available: https://git-scm.com/book/en/v2/Git-Tools-Submodules.

[10] “Azure Terraform library,” [Online]. Available: https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/network_security_group.

[11] “Apache 2.0 License,” [Online]. Available: https://www.apache.org/licenses/LICENSE-2.0.

APPENDIX: Implementation, delivery and usage

1 Implementation

1.1 Fitting into overall PIACERE Architecture

The role of the Infrastructural Code Generator is the same defined in the deliverable D3.4 [7]. During this third iteration, we refine the ICG API, and extend the output information to add custom IaC code provided by the user.

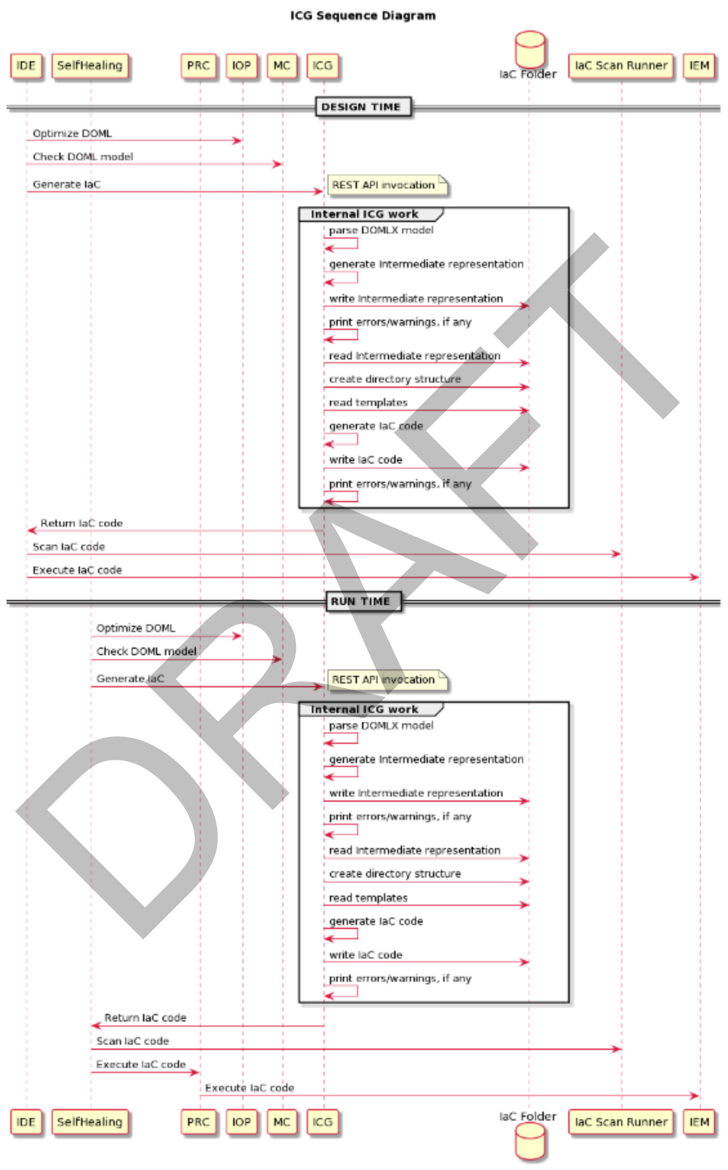


Figure 2: ICG sequence diagram

Figure 2 sequence diagram shows the ICG interaction with the other PIACERE components. Its behaviour has slightly changed from the previous deliverable D3.5 [1] in the runtime section: the PIACERE SelfHealing calls ICG instead of the PRC; it still sends to the ICG the DOML model, plus in this version the optional user's asset folder, and requests the generation of the IaC code. The ICG packs everything in a zip folder. The outcome is then security checked by the IaC Scan Runner and is executed by the IaC Executor Manager (IEM).

The ICG receives as input the DOML model which can be either completely defined by the user or optimized by the IaC Optimizer Platform (IOP).

The generated IaC files include code for the PIACERE internal components, in particular in this new version the security agents IaC files are produced next to the monitoring agents IaC files.

The main updates introduced in this new version is the interaction with the PIACERE IDE: in the previous version the IDE sent the DOMLx model to the ICG and received the zip folder as output. Now, the IDE sends to the ICG a zip folder with the DOMLx model, the DOML ecore file and the user's asset folder. In the output zip folder we added the user's asset, IaC files for configuring security monitoring and the Gaia-X self-description.

1.2 Technical description

This section describes the technical details of the implemented ICG software for the M30.

1.2.1 Prototype architecture

The Infrastructural Code Generator is a microservice application working both from command line and via REST API. It is invoked through the REST API and takes as input the DOML model in the XML format for the generation of the Infrastructure as Code (IaC) which is returned in an archive (zip) format. The internal architecture now includes a new plugin implementing the new target language Docker Compose.

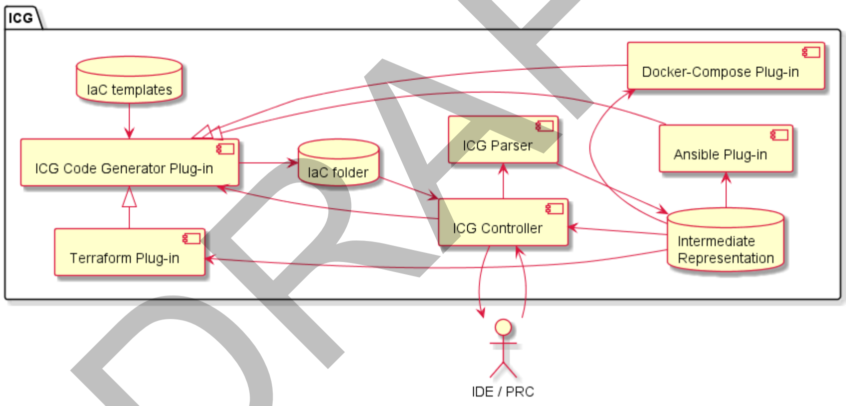


Figure 3: ICG internal architecture

Figure 3 shows the ICG architecture in this new version. In order to understand better the ICG functionalities, we summarize here the main internal components scope.

ICG Controller is the component responsible for the communication with the external tools. It exposes REST API and is called by the PIACERE IDE and PIACERE PRC. It redirects the requests to the ICG Code Generator Plug-in.

ICG Parser is is activated by the Controller and reads the input model in XML format, a DOML representation that is called DOMLx, to produce an Intermediate Representation compatible with the ICG plug-ins. The last version of the Parser is compatible with DOML v3.

Intermediate Representation (IR) is a JSON file which describes the infrastructure to be generated. Its syntax has been described in deliverable D3.4 [7], section 2.2.2.3.

Terraform Plug-in is the component responsible for the Terraform IaC files generation. It reads all the information needed for the creation of the infrastructure from the IR.

Ansible Plug-in is the component responsible for the Ansible IaC files generation. It reads all the information needed for the creation of the infrastructure from the IR.

Docker Plug-in is the new component responsible for the Docker Compose IaC files generation. Like the other plug-ins, it reads all the information needed for the creation of the infrastructure from the IR.

IaC Templates Library contains the templates used for the IaC files and the metadata information. This library in this new ICG version has been extended to support new resources, providers and plug-ins.

The **IaC folder** is an ephemeral storage to host the IaC files and configuration produced during the generation process. This folder is be the output of the ICG and will be described in Section 1.2.3.

The interaction between these components is not changed and is represented in the sequence diagram shown in Figure 4.

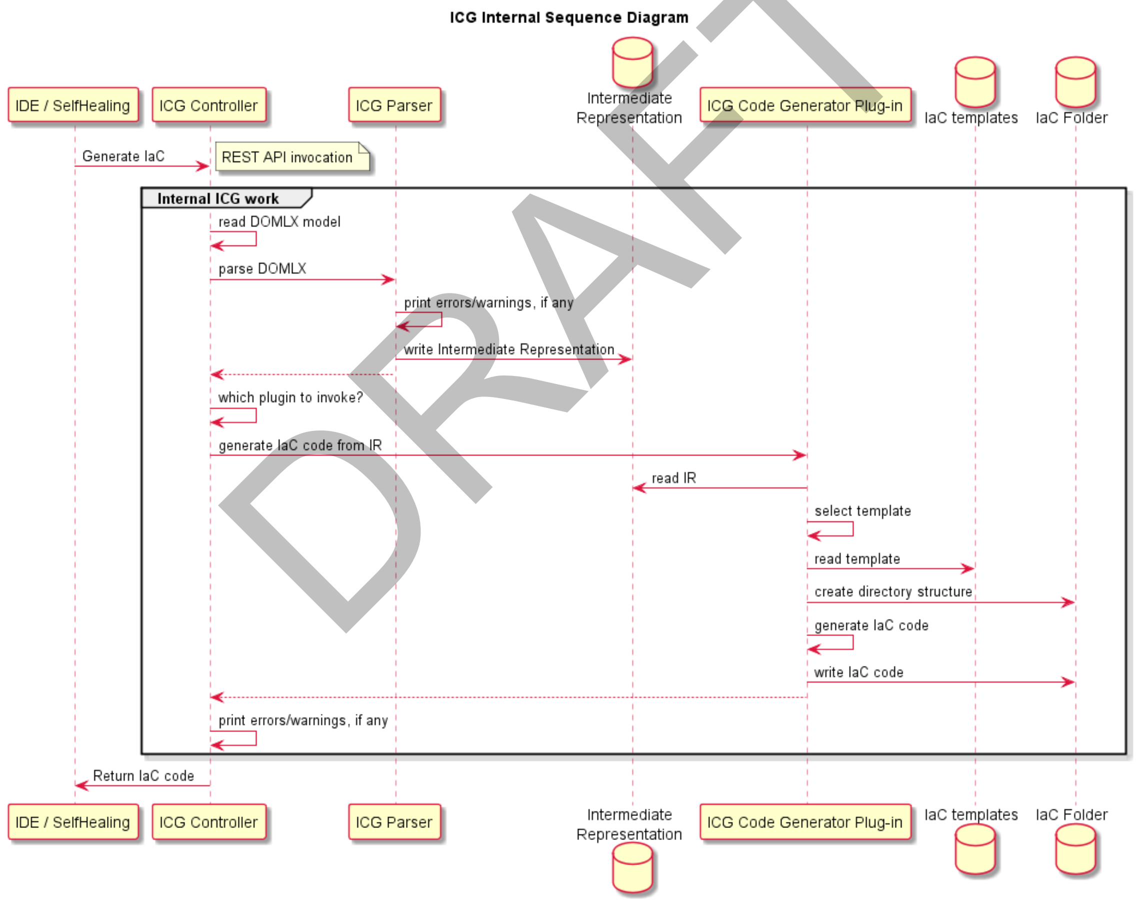


Figure 4: ICG Internal Sequence Diagram

1.2.2 Components description

The ICG components have not changed their scope, but some updates and extensions have been introduced. In this version a new plug-in is introduced for the generation of Docker Compose, the ICG Controller and ICG Parser are updated to support a new PIACERE functionality and the template library is extended now with new templates.

1.2.2.1 ICG Controller

A new functionality has been introduced in the PIACERE Framework. The SIMPA Use Case indeed proposed to give the user the possibility to upload some extra IaC code implemented by the user themselves. There are situations in which IaC code was implemented by developers in the past and should be reused by importing it into the PIACERE project.

For this reason, the IDE evolved and allows the user to put beside the DOML definition another folder containing external IaC code. These external files are going to be part of the IaC files generated by the ICG and will follow the same PIACERE workflow: the IaC Scan Runner is going to scan them for security issues and the IEM is going to execute them.

Thus, the ICG Controller exposes a new REST API which is a POST request at the "/iac/files/upload" and takes as input the zip folder containing: the DOML ecore file, the DOML model in XML format (called DOMLx) and the user's extra folder.

The ICG Controller then redirects this information to the ICG Parser and the ICG Generator Plug-in, receives the generated IaC files and archive them with the user's folder into a single zip. This zip file is sent as output.

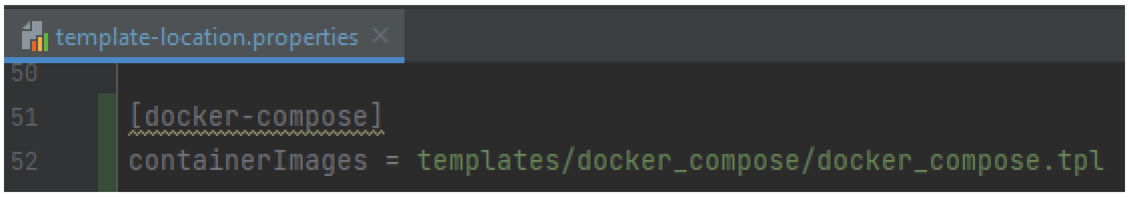
1.2.2.2 ICG Parser

The ICG Parser is the component responsible for the navigation and translation of the DOML model into the Intermediate Representation (IR) JSON file, used by the ICG as source of truth for the templates generation (see deliverable D3.4 [7] for further information).

In this new version the ICG Parser is evolved, it now generates information for new external IaC language. The ICG indeed supports extension mechanisms which allow the user to introduce new templates and IaC languages, as described in Sections 2.3.3 and 2.3.4.

This component now, after parsing the DOML model to generate the IR for the default Terraform and Ansible languages, reads the *template-location.properties* file and checks if new IaC languages have been introduced. In the case the ICG Parser finds one or more of them, it parses again the DOML model adding new steps for the new resources to be generated.

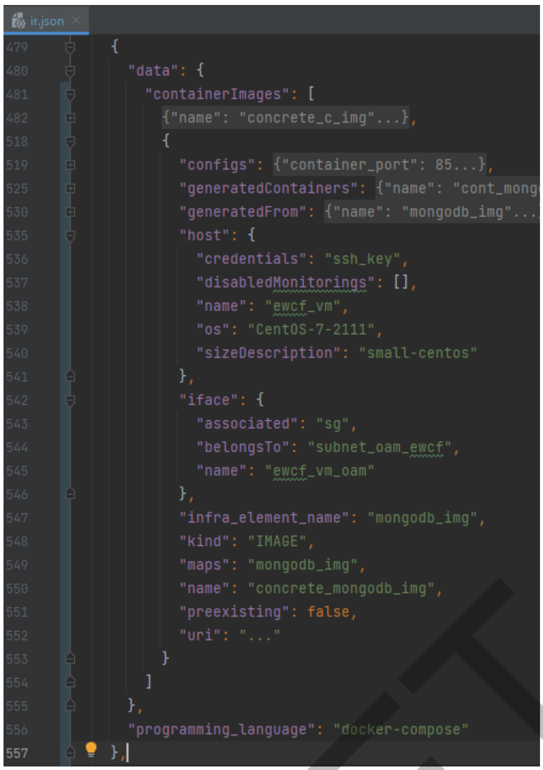
Let's assume for example that the new "docker-compose" IaC language is introduced into the *template-location.properties* file and that this language is responsible for the creation of the DOML ContainerImages resources, as shown in figure below.



```
[docker-compose]
containerImages = templates/docker_compose/docker_compose.tpl
```

Figure 5: ICG Parser - new IaC language

The ICG Parser reads this information, find out the new "docker-compose" language and navigate the DOML model to find out all the information related to the containerImages resources. The results is a new step into the IR:



```
{
  "data": {
    "containerImages": [
      {"name": "concrete_c_img"...},
      {
        "configs": {"container_port": 85...},
        "generatedContainers": {"name": "cont_mong
        "generatedFrom": {"name": "mongodb_img"...
        "host": {
          "credentials": "ssh_key",
          "disabledMonitorings": [],
          "name": "ewcf_vm",
          "os": "CentOS-7-2111",
          "sizeDescription": "small-centos"
        },
        "iface": {
          "associated": "sg",
          "belongsTo": "subnet_oam_ewcf",
          "name": "ewcf_vm_oam"
        },
        "infra_element_name": "mongodb_img",
        "kind": "IMAGE",
        "maps": "mongodb_img",
        "name": "concrete_mongodb_img",
        "preexisting": false,
        "uri": "..."
      }
    ]
  },
  "programming_language": "docker-compose"
},
```

Figure 6: ICG Parser - new step for new IaC language

1.2.2.3 Docker-Compose Plugin

A new plugin for Docker Compose is now supported by this version of the ICG to implement the deployment DevOps phase. This plugin is responsible for the generation of the Docker Compose IaC files and has been introduces as an external plug-in, as described in Section 2.3.4.

1.2.2.4 Template Library

During this last period of development the available templates library was expanded, potentially more than what was done for the previous two deliverables. Following in this chapter is a brief description of these new templates.

First were added new templates for resources that didn't need new definitions in DOML, in this category there are:

- The Ansible templates to configure the ElasticSearch application on Ubuntu and CentOS operating systems. This application needed multiple templates for a single application, so the ICG logic was updated to be capable of handling multiple templates for one application.
- The Terraform template to manage security groups on the Azure provider.

As for the new templates that required more complex implementations, first there were the resources that required new definitions inside DOML:

- The first resource in this category is the autoscaling group, added for the UC2 and for the AWS provider in the beginning, this resource required the implementation of the autoscaling resource inside DOML. For this, resource specific Terraform templates were added. Being now implemented as a known resource the addition of the templates for this resource to the other providers is now simplified.

- The second resource in this category is the generic resource, this concept was added in DOML as a requirement of UC1 and it allows to easily extend DOML. The generic resource can be used to model all those resources that are required for specific cases and are not yet available in DOML. To manage the generic resource the ICG has added the related logic and some templates. Some of the templates that use the generic resource are the Terraform templates for VMware: data_resources.tpl; datacenter.tpl; datastore.tpl.

Another category is made of the templates produced for the addition of new providers; during this development period, the following two new providers were added:

- VMware: added as a requirement of the UC1; for this provider, multiple templates have been produced for the deployment of VMs, networks and storage.
- IONOS: added as an integration related to the Gaia-X collaboration; for this provider, all the templates necessary for the deployment of VMs have been produced.

Separately, the code and templates for the configuration of the performance and security monitoring agents were refactored. One of the relevant changes derived from this refactoring being the introduction of cross-platform templates for Ansible, for the IaC that can deploy applications on all the operating systems.

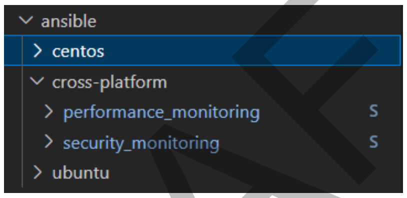

```
ansible
  centos
  cross-platform
    performance_monitoring
    security_monitoring
  ubuntu
```

Figure 7: New cross-platform folder

Last category of templates added is all the templates produced to support the addition of the docker compose IaC language. As previously stated in chapter [3.3] this IaC language was added as a requirement of UC3, to support this new IaC language multiple templates were generated such as main.tpl and docker-compose.tpl.

1.2.3 Output description

The ICG output is a zip folder with the IaC files generated and the instructions on how execute them.

The folders structure defined for the output is not changed from the previous iteration but new files are now available.

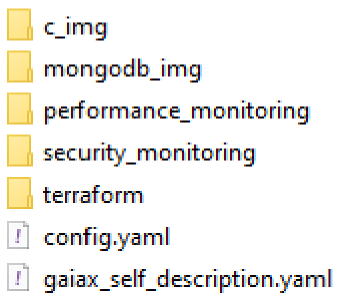

```
c_img
mongodb_img
performance_monitoring
security_monitoring
terraform
config.yaml
gaiax_self_description.yaml
```

Figure 8: ICG output

The integration with the PIACERE security agents is completed and the result is a new dedicated folder called "security_monitoring". Furthermore, a new file called "gaiax_self_description.yaml" is now created in the root folder and used for the integration with the Gaia-X European initiative [5].

1.2.4 Technical specifications

The ICG is still a microservice application implemented with Python version 3.6 and using the 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

In this new version two more libraries have been introduced to support the new APIs specification. The input zip folder taken as input indeed is uploaded in multiparts and using an asynchronous method.

- Aiofiles version 23.1.0
- Python-Multipart version 0.0.6,

Other libraries used by ICG are the following:

- Fast API version 0.74.1 and Uvicorn version 0.17.5 for the REST API implementation
- PyEcore version 0.12.2 for the ICG Parser
- PyYaml version 6.0 for the creation of the configuration files given in the output compressed folder

2 Delivery and usage

2.1 Package information

During this period we finalized the integration with the PIACERE internal tools, we added a new plugin and its relative templates. This result in the following added folders with respect to the previous deliverable:

- **ansible/cross-platform** folder is a new folder extending the template library. It is a GitLab submodules redirection which link this folder to the PIACRE monitoring and security agents code.
- **docker-compose** folder in the template library is a new directory with the new templates to be generated for this plugin.

The folders and files most useful to the user for the setup and management of the ICG are the following:

- **Templates** folder: contains the templates to be used for the IaC code generation grouped by the IaC language
- **Template-location.properties** file: this file lists the reference to the proper template to be used for the generation of a specific DOML resource
- **Input_file_generated** folder: contains the Intermediate Representation generated by the ICG
- **Dockerfile** and **requirements.txt** files: used for the Docker packaging of the application
- **Doc** folder: contains some example scenarios of usage of the ICG

The ICG components are organized in packages and folders containing these packages are the following:

- **Controller**: folder dedicated to the ICG Controller package
- **Icgparser**: folder dedicated to the ICG Parser package
- **Plugin**: folder containing the packages of the Terraform and the Ansible ICG plug-ins

2.2 Installation instructions

The installation of this ICG version is not changed with respect to the one described in the previous deliverable except for the download of the code from the GitLab repo.

Indeed, the ICG is responsible for the code generation of the PIACERE internal tools, in particular it takes the source code developed for the monitoring and security agents and adds some extra information and structure hierarchy in order to be executed correctly by the IEM. These PIACERE agents code is stored in a separated git repository. For this reason, the ICG uses the Git submodules functionality [9], which allows the external code to be linked and automatically refreshed.

The ICG code and submodules can be downloaded from the public TECNALIA GitLab repository through the following command:

```
git clone https://git.code.tecnalia.com/piacere/public/the-platform/icg

git submodule update --init --recursive
```

The ICG can still be executed both as a command line tool and as a docker-container and the next steps for its installation are the following.

Once the code is downloaded, you can run it with docker executing the following commands from the root of the project:

```
docker build -t icg:1.0.0 .
docker run --name icg -d -p 5000:5000 icg:1.0.0
```

and finally the API docs are available at: http://localhost:5000/docs.

The ICG command line is still available and the tool can be executed with Python 3.6 too as described in D3.4 [7], installing Jinja2 v3.0.3 library as in the previous version and including also PyYAML v6.0, FastAPI v0.74.1, Uvicorn v0.17.5 and PyEcore 0.12.2 libraries. The following command can be used to install all these libraries:

```
pip install -r requirements.txt
```

As previously introduced the ICG can be run as a command line tool, an example of how it can be done on Windows is the following:

```
py .\main.py -d icgparser/doml --single icgparser/doml/nginx-openstack_v2.domlx
```

2.3 User Manual

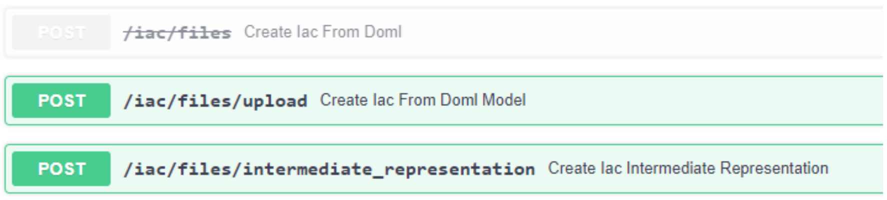
The user can run the ICG both from command-line and from docker container, as described in the previous section. Once the application is up and running, it can be used through REST API or command line.

2.3.1 ICG usage from REST API

The ICG API are implemented with the Fast API library and exposed at the /docs endpoint, e.g. if the application is running on a local environment they can be found at http://localhost:5000/docs. The REST APIs are reachable also from the Internet at https://icg.ci.piacere.digital.tecnalia.dev/docs.

The ICG REST API to create the IaC files is updated to support the possibility for the user to add his/her own IaC files. The previous POST "/iac/files" API is deprecated and substituted by a POST endpoint "/iac/files/upload". The API takes as input a zip folder with the doml model, the ecore file and the optional asset folder with the user's IaC external files. The output is a zip folder with the IaC generated.

A new POST API is introduced to help the user for the template generation: the input is the zip folder with the doml model, the ecore file and the user's folder and the output is the Intermediate Representation generated by the ICG.


POST ~~/iac/files~~ Create Iac From Doml

POST /iac/files/upload Create Iac From Doml Model

POST /iac/files/intermediate_representation Create Iac Intermediate Representation

Figure 9: ICG REST API

The user can test these APIs from the GUI page in an easy way. For example, to test the /iac/files/upload endpoint the user can:

- Press “try it out” button on the top right
- Press “choose file” and select the zip file with the doml model, ecore and assets folder
- Press “execute”

The output is a link to download the file.

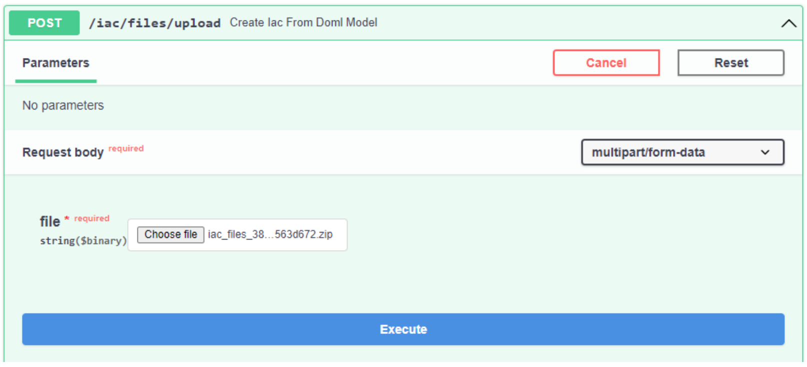
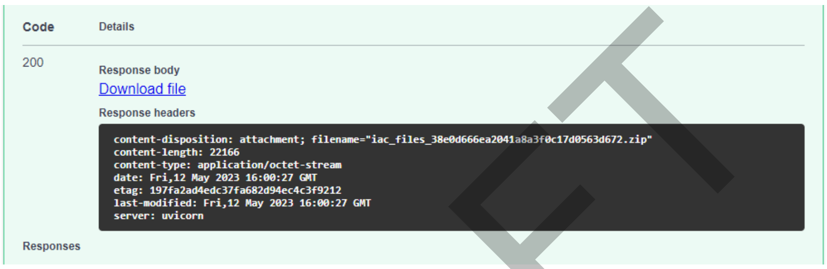


Figure 10: ICG POST /iac/files/upload example



```
content-disposition: attachment; filename="iac_files_38e0d666ea2041a8a3f0c17d0563d672.zip"
content-length: 22166
content-type: application/octet-stream
date: Fri,12 May 2023 16:00:27 GMT
etag: 197fa2ad4edc37fa682d94ec4c3f9212
last-modified: Fri,12 May 2023 16:00:27 GMT
server: uvicorn
```

Figure 11: ICG POST /iac/files/upload example output

2.3.2 ICG usage from command line

The ICG usage from command line is not changed from the description in the previous deliverable a part from the usage of the curl command to call the APIs.

The generation of the IaC code, assuming the ICG runs on http://127.0.0.1:5000 and the input folder name is *doml_model_ecore_assets.zip*, is:

```
curl -X 'POST' \
  'http://127.0.0.1:5000/iac/files/upload' \
  -H 'accept: application/json' \
  -H 'Content-Type: multipart/form-data' \
  -F 'file=@doml_model_ecore_assets.zip;type=application/x-zip-compressed'
```

The generation of the Intermediate Representation instead can be request as follows:

```
curl -X 'POST' \
  'http://127.0.0.1:5000/iac/files/upload' \
  -H 'accept: application/json' \
  -H 'Content-Type: multipart/form-data' \
  -F 'file=@doml_model_ecore_assets.zip;type=application/x-zip-compressed'
```

2.3.3 ICG extension: writing a new template

Allowing for easy extensibility is an important requirement of the ICG. During the development, the focus was on allowing three types of extensibility: adding a component by adding one template; adding a new provider; adding a new target language. For the extension to work end-to-end, it needs to be also aligned with an extension on the DOML side, this can be achieved taking advantage of the DOML extension mechanisms.

The first and simplest kind of extension is adding a new component by adding a new template. To write a functioning template there are a few rules that is useful to know and, in this chapter, we'll go through them.

The templates follow a Jinja2 templatization model so they follow all the established rules by Jinja2 standard. The common process for writing a new template is by tacking a known working IaC file or definition, extrapolate the relevant variables that can be parametrized or can be expanded through the addition of logic, finally using the Jinja2 standards to define the parameters.

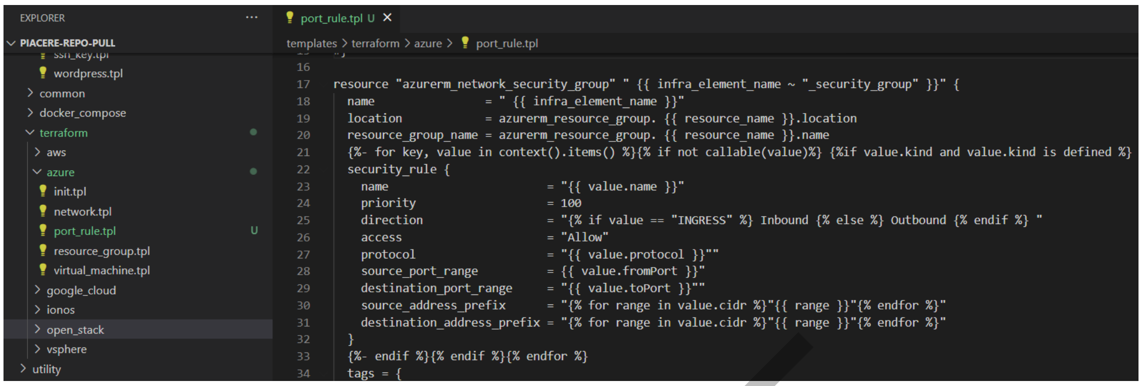
Let's follow an example to better understand all this process, in this example we'll add the security group resource for the Azure provider. This is an infrastructure layer resource and will be generated through Terraform, so first come de definition of the related IaC, for this example we'll use the example provided by the Terraform library [10]. Below is the original code used as a reference.

```
resource "azurerm_network_security_group" "example" {
  name                = "acceptanceTestSecurityGroup1"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name

  security_rule {
    name                       = "test123"
    priority                   = 100
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "*"
    source_address_prefix      = "*"
    destination_address_prefix = "*"
  }

  tags = {
    environment = "Production"
  }
}
```

Now from this template we can identify the variables that can be parametrized, such as the name, the direction or the protocol. Let's see how the end template looks after this work.

```
resource "azurerm_network_security_group" " {{ infra_element_name ~ "_security_group"
}}" {
  name                = " {{ infra_element_name }}"
  location            = azurerm_resource_group. {{ resource_name }}.location
  resource_group_name = azurerm_resource_group. {{ resource_name }}.name
  {%- for key, value in context().items() %}{% if not callable(value)%} {%if value.kind
and value.kind is defined %}
  security_rule {
    name                       = "{{ value.name }}"
    priority                   = 100
    direction                  = "{% if value == "INGRESS" %} Inbound {% else %}
Outbound {% endif %} "
    access                     = "Allow"
    protocol                   = "{{ value.protocol }}""
    source_port_range          = {{ value.fromPort }}"
    destination_port_range     = "{{ value.toPort }}""
    source_address_prefix      = "{% for range in value.cidr %}"{{ range }}"{% endfor
%}"
    destination_address_prefix = "{% for range in value.cidr %}"{{ range }}"{% endfor
%}"
  }
  {%- endif %}{% endif %}{% endfor %}
  tags = {
    environment = "Production"
    }
}
```

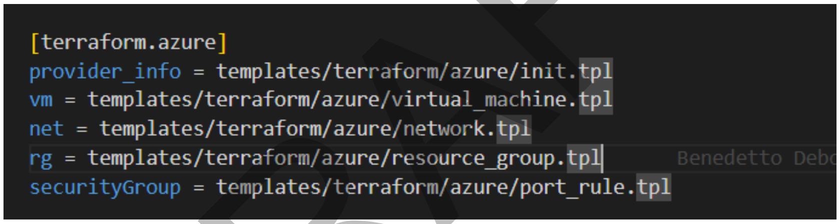
Once the template is ready the next step is to add it inside the correct folder, in our case the template is for terraform, for the provider Azure so that is going to be the path. The file was named port_rule.tpl, the name is to the user discretion.



```

resource "azurerm_network_security_group" " {{ infra_element_name ~ "_security_group" }}" {
  name                = " {{ infra_element_name }}"
  location            = azurerm_resource_group. {{ resource_name }}.location
  resource_group_name = azurerm_resource_group. {{ resource_name }}.name
  {%- for key, value in context().items() %}{% if not callable(value)%} {%if value.kind and value.kind is defined %}
  security_rule {
    name                       = "{{ value.name }}"
    priority                   = 100
    direction                  = "{% if value == "INGRESS" %} Inbound {% else %} Outbound {% endif %} "
    access                     = "Allow"
    protocol                   = "{{ value.protocol }}""
    source_port_range          = {{ value.fromPort }}"
    destination_port_range     = "{{ value.toPort }}""
    source_address_prefix      = "{% for range in value.cidr %}"{{ range }}"{% endfor %}"
    destination_address_prefix = "{% for range in value.cidr %}"{{ range }}"{% endfor %}"
  }
  {%- endif %}{% endif %}{% endfor %}
  tags = {
```

Figure 12: New generated template

For the ICG to be aware that this template is available it is also important to add the specification inside the template-location.properties file like in the figure below.



```
[terraform.azure]
provider_info = templates/terraform/azure/init.tpl
vm = templates/terraform/azure/virtual_machine.tpl
net = templates/terraform/azure/network.tpl
rg = templates/terraform/azure/resource_group.tpl
securityGroup = templates/terraform/azure/port_rule.tpl
```

Figure 13: New templates in ICG properties file

With this, the new template is added to the ICG, but as specified in the beginning of the chapter for the end user to use this new resource it's necessary to add it also inside the DOML model. It is also important that the parameters defined inside the ICG have a corresponding property inside DOML.

Let's have a fast look also on how to do this to understand the end-to-end process, still the implementation on the DOML will be more detailed in the next Doml-E deliverable D3.3, which is going to be released at M30. In DOML resources can be added from the already available for example for a new provider like in our example, alternatively entire new resources can be added using the generic resource concept. Once the resource is added inside the DOML, the last step is to add in the properties field all the variables present inside the template, for the correct association it is enough that the key used for these variables is the same inside DOML and the template.

2.3.4 ICG extension: adding a new target language

The ICG extension mechanism provides the possibility to add a new target language. This feature corresponds to do the following actions:

- Add templates for the new target language in the template library, as described in Section 2.3.3

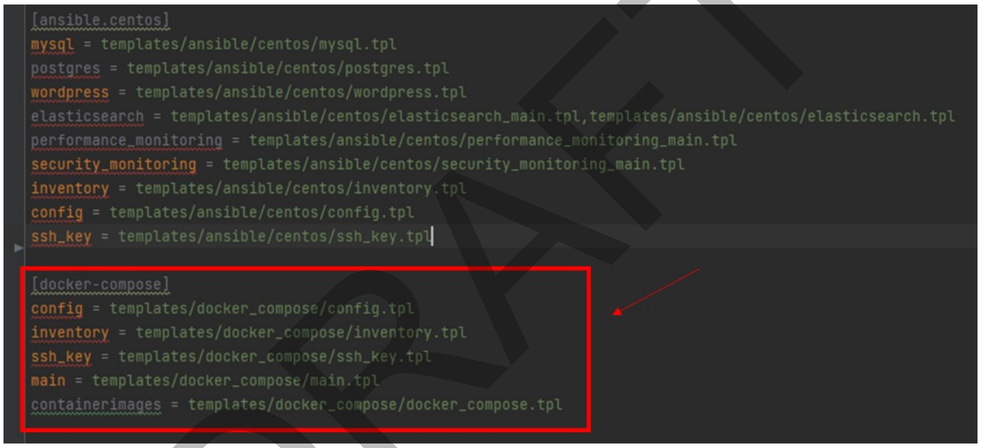
- Update the *template-location.properties* files according to the new templates and provider
- Add a new plugin class for the new target language templates generation

The *template-location.properties* is the source of truth for the ICG to know what are the target languages available. In the properties files there are different sections, each one corresponding to an IaC language, and for each section there is a list of resources created with that specific language.

The ICG supports by default the Ansible and Terraform IaC languages, but the user can add a new section with the new target language in the properties files and a new plugin class for their generation.

An example of a new plugin introduced in this new version of the ICG using the extension mechanism is the Docker Compose Plugin, which is the responsible IaC language generator for docker compose.

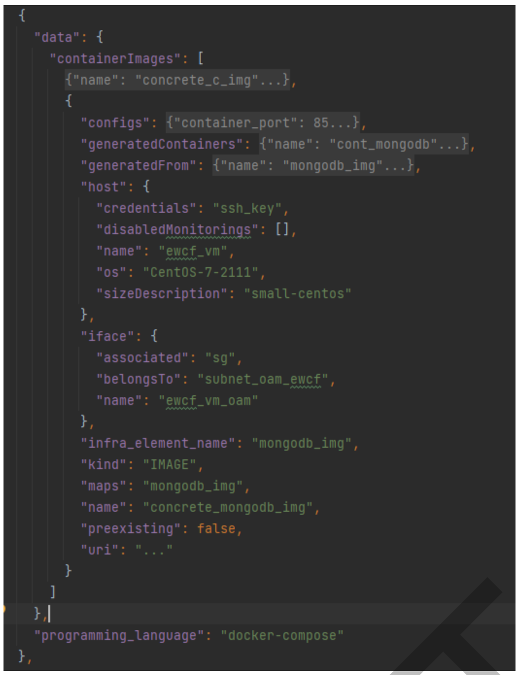
First of all, the *template-location.properties* is updated adding the new "docker-compose" section and the resources and templates it is responsible for, as shown in the figure below.



```
[ansible.centos]
mysql = templates/ansible/centos/mysql.tpl
postgres = templates/ansible/centos/postgres.tpl
wordpress = templates/ansible/centos/wordpress.tpl
elasticsearch = templates/ansible/centos/elasticsearch_main.tpl,templates/ansible/centos/elasticsearch.tpl
performance_monitoring = templates/ansible/centos/performance_monitoring_main.tpl
security_monitoring = templates/ansible/centos/security_monitoring_main.tpl
inventory = templates/ansible/centos/inventory.tpl
config = templates/ansible/centos/config.tpl
ssh_key = templates/ansible/centos/ssh_key.tpl

[docker-compose]
config = templates/docker_compose/config.tpl
inventory = templates/docker_compose/inventory.tpl
ssh_key = templates/docker_compose/ssh_key.tpl
main = templates/docker_compose/main.tpl
containerimages = templates/docker_compose/docker_compose.tpl
```

Figure 14: ICG Extension - new docker-compose section

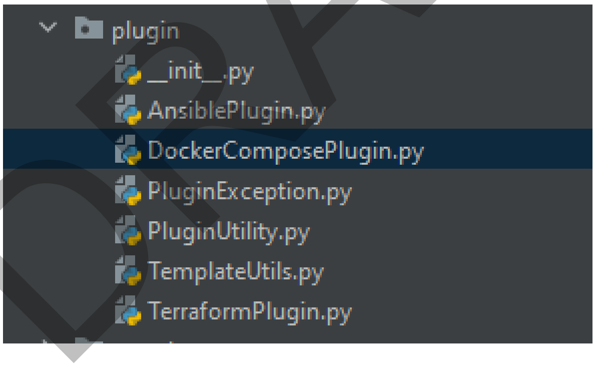
The ICG scans the *template-location.properties* file and found out a new section for "docker-compose". It reads the resources this section takes care of and introduces into the Intermediate Representation their information extracted from the DOML model.



```
{
  "data": {
    "containerImages": [
      {"name": "concrete_c_img"...},
      {
        "configs": {"container_port": 85...},
        "generatedContainers": {"name": "cont_mongodb"...},
        "generatedFrom": {"name": "mongodb_img"...},
        "host": {
          "credentials": "ssh_key",
          "disabledMonitorings": [],
          "name": "ewcf_vm",
          "os": "CentOS-7-2111",
          "sizeDescription": "small-centos"
        },
        "iface": {
          "associated": "sg",
          "belongsTo": "subnet_oam_ewcf",
          "name": "ewcf_vm_oam"
        },
        "infra_element_name": "mongodb_img",
        "kind": "IMAGE",
        "maps": "mongodb_img",
        "name": "concrete_mongodb_img",
        "preexisting": false,
        "uri": "..."
      }
    ]
  },
  "programming_language": "docker-compose"
},
```

Figure 15: ICG Extension – Intermediate Representation with docker-compose resources

The Intermediate Representation is the file the ICG uses to choose the plugin language for the IaC generation and contains the resources information for the template's creation. The new step of the Intermediate Representation is sent to the Docker Compose Plugin. This Plugin is a python class added by the user into the plugin library.



```
plugin
    __init__.py
    AnsiblePlugin.py
    DockerComposePlugin.py
    PluginException.py
    PluginUtility.py
    TemplateUtils.py
    TerraformPlugin.py
```

Figure16: ICG Extension – docker-compose plug in

2.4 Licensing information

The ICG component has been released in open source under the Apache 2.0 license [11].

2.5 Download

The ICG source code is still available on public repositories and can be downloaded from the public TECNALIA GitLab repo at

https://git.code.tecnalia.com/piacere/public/the-platform/icg or from the public HPE GitHub

repo at: https://github.com/HewlettPackard/icg-iac-code-generator .