PIACERE

Deliverable D3.3

PIACERE Abstractions, DOML and DOML-E – v3

Editor(s):	Sergio Canzoneri, Elisabetta Di Nitto
Responsible Partner:	Politecnico di Milano/ PoliMi
Status-Version:	Final - v1.0
Date:	31.05.2023
Distribution level (CO, PU):	Public

Project Number:	101000162
Project Title:	PIACERE

Title of Deliverable:	PIACERE Abstractions, DOML and DOML-E – 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):	Politecnico di Milano/PoliMi
Contributor(s):	Go4it, HPE, Prodevelop, Tecnalia
Reviewer(s):	Ismael Torres (Prodevelop)
Approved by:	All Partners
Recommended/mandatory readers:	WP4, WP5, WP6, WP7

Abstract:	This deliverable is the output of tasks 3.1, 3.2 and 3.3. It presents the final version of the DOML (v3.0). DOML is a domain-specific language designed for modelling the cloud applications and the infrastructural resources, hiding the specificities and technicalities of the current IaC solutions and increases the productivity of these teams. DOML is complemented by DOML-E (KR4), which is the set of extension mechanisms defined for the language. They allow new infrastructural components, e.g., for software execution, network communication, cloud services, or data storage, to be incorporated in the DOML language. This deliverable presents the DOML metamodel and syntax and its extension mechanisms, DOML-E. Moreover, it includes an overview of the changes in the language since the previous deliverable D3.2 and discusses about the level of accomplishment of the requirements formulated within the PIACERE project. A preliminary evaluation of DOML is presented together with some examples of DOML usage. Finally, the deliverable includes an overview about the lessons learnt and future directions for the extension of the language beyond the scope of the PIACERE project.
Keyword List:	Model-driven engineering, metamodels, modelling abstractions, 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	16.03.2023	Table of contents defined	PoliMi
v0.2	06.04.2023	Material collected	ALL partners
v0.3	15.05.2023	Complete draft version ready for review	PoliMi
v.0.4	19.05.2023	Reviewed version	Prodevelop
v0.5	29.05.2023	Final version after review.	PoliMi
v1.0	30.05.2023	Final quality check. Ready for submission	TECNALIA

Table of contents

List of tables

List of figures

Terms and abbreviations

CMS	Content Management System
CSP	Cloud Service Provider
DevOps	Development and Operation
DoA	Description of Action
EC	European Commission
EDMM	Essential Deployment Metamodel
FaaS	Function as a Service
GA	Grant Agreement of the project
IaC	Infrastructure as Code
ICG	IaC Code Generation
ICMP	Internet Control Message Protocol
IEP	IaC Execution Platform
IOP	IaC Optimization
KPI	Key Performance Indicator
MC	Model Checker
NFR	Non-Functional Requirement
SW	Software
TBCG	Template-Based Code Generation
VM	Virtual Machine
AWS	Amazon Web Services

Executive Summary

This document extends deliverable D3.2 [1], including the progress of the DOML development to fulfil the requirements during the evolution of the project.

DOML (PIACERE KR1) is a domain-specific language designed for modelling cloud applications and infrastructural resources, hiding the specificities and technicalities of current IaC solutions and increasing productivity of these teams. DOML models are created using the PIACERE IDE (PIACERE KR2), which provides users with guidance and it also integrates all other design-time PIACERE tools. Then, the DOML models are translated through the *Infrastructural Code Generator* (ICG, PIACERE KR3), into the target IaC languages for complex applications.

The DOML is complemented by DOML-E (KR4), which is a set of extension mechanisms defined for the language. They allow new infrastructural components, e.g., for software execution, network communication, cloud services, or data storage, to be incorporated in the DOML language.

This deliverable provides an overview of DOML and DOML-e. Furthermore, it highlights the changes that have been introduced in DOML compared to what was reported in the previous deliverables and presents the status of requirements fulfilment. Additionally, the document highlights the main innovations introduced by DOML and DOML-e and provides an overview of the experiments developed so far by using the DOML. Finally, it presents the main lessons learnt, an outlook to the future beyond the scope of the PIACERE project, and the conclusions of this deliverable. It includes in the Appendix the DOML details for one of the PIACERE Use Cases and a guide on how to extend DOML.

The deliverable is also accompanied by two external annexes, the first one [2] provides a detailed definition of all concepts of the DOML, the second one (Canzoneri, 2023) is a tutorial that allows end users to become proficient with the language. Both are released as separated documents to facilitate their usage and evolution independently of this deliverable.

1 Introduction

This deliverable presents the DOML language and the mechanisms that have been developed to support its extension. It is an update of the previous deliverable D3.2 [1] but it is written to be self-contained.

Through the PIACERE project development, multiple versions of the DOML language have been released, starting from DOML 1.0 presented in deliverable D3.1 [4], to DOML 3.0 which is the focus of the current deliverable.

DOML is characterised by a metamodel, encoded in ECore [5]-- a metamodeling approach offered by Eclipse -- and by a syntax. Thanks to the usage of Xtext [6], the DOML editor guides the user through the DOML syntax and allows him/her to define syntactically correct models. The DOML-DOMLX conversion is a supporting service that, given a DOML model can generate a serialized representation of its data structures, the DOMLX, which is the DOML PIACERE internal representation. By the end of the project, the DOML-DOMLX conversion service will support also the backward translation from DOMLX into the DOML syntax. Thanks to DOMLX, the PIACERE tools can maintain their independency from the specific syntax adopted for the DOML. Moreover, the backward translation that will be offered by the DOML-DOMLX conversion service will allow even the transformation of DOML models written according to a previous version of the language into the latest version, thus greatly contributing to the backward compatibility of the language.

1.1 About this Deliverable

The purpose of this deliverable is to provide a general presentation of the DOML and its extension mechanisms, DOML-E.

As mentioned above, the main objective of the DOML has been to reduce the effort needed to automate the deployment and operation of an application in combination with its underlying infrastructure. This has resulted in the development of a high-level modeling language that is then translated, through the ICG, into de-facto IaC standard languages supporting the Ops phases of the software lifecycle.

The DOML has been developed taking as a reference the resources and configurability options made available by the main cloud providers, considering the main characteristics of the IaC languages used as a reference, and taking into account the needs of the PIACERE case studies.

The purpose of this deliverable is to report on the work done, to show in practice how DOML works and how it can be used in concrete examples, and to provide a summary of lessons learnt and the plan for future development beyond the end of the PIACERE project.

Being this the final version of a series of three deliverables, it has been chosen to make it self-contained. As such, part of its content is a repetition and revision of what has been presented in previous deliverables.

The main innovations introduced since the previous version of this deliverable D3.2 [1], consist in the following aspects:

- Consolidation and clean-up of the DOML modelling language which has now reached version 3.0. Through the PIACERE project the following main versions of the language have been released: 1.0 at the end of the first project year, 2.0 and 2.1 during the second project year, 2.2.1 and 2.2.2 in the third year.
- Consolidation of the extension mechanisms (DOML-E).

- Elaboration of the examples of use of the DOML language using the PIACERE IDE, their validation through the Model Checker, their optimization through the IOP tool, and the generation of the corresponding IaC code through the ICG.

As discussed in further details in this deliverable, of the 18 requirements defined in PIACERE and associated to the DOML, 16 have been achieved completely and 2 partially.

1.2 Document Structure

The document is organized as follows:

- Section 2 provides a general overview of the DOML and DOML-e, of their changes compared to the version reported in D3.2, and of their main innovations.
- Section 3 provides an overview of the examples of usage of the DOML and DOML-e.
- Section 4 summarizes the lessons learnt and the plan for future development.
- Section 5 concludes the deliverable.

The deliverable is accompanied by an Appendix providing further details on a DOML model example that has been used as a reference to extend the language and on how to extend the DOML language by introducing new concepts in the DOML metamodel. Finally, the deliverable includes also Annex 1 [2] which presents the detailed specification of the DOML concepts and Annex 2 (Canzoneri, 2023) which provides a tutorial of the language usage.

2 DOML (KR1) and DOML-E (KR4)

2.1 Overview

The DevOps Modelling Language (DOML) aims at offering a high-level declarative approach to the definition of an application and its infrastructure. DOML models are mainly structured in three layers. Specifically, software components (e.g., web servers, databases, etc.) are described in the *application layer*, abstracting away from the infrastructure on which they are supposed to run. Infrastructure components are specified in the *abstract infrastructure layer*, and then linked to the applications they are supposed to host. This layer models infrastructural facilities, such as virtual machines, networks, containers, etc., without referring to their actual concretization in specific technologies (e.g., AWS or OpenStack VMs, Docker containers). This last aspect is tackled by the *concrete infrastructure layer*, where the user specifies the infrastructure components offered by the Cloud Service Provider (CSP).

The adoption of this three layers approach allows us to overcome a limitation of the currently available Infrastructure as Code (IaC) languages that tend to focus on specific aspects, either the configuration of the specific infrastructural elements or the installation and configuration of software elements, thus making difficult for DevOps teams to have a complete overview of their whole system [7].

As it happens for typical programming languages, DOML offers some extension mechanisms, DOML-E, that keep the language open to the addition of new resources, both at the abstract and concrete layers.

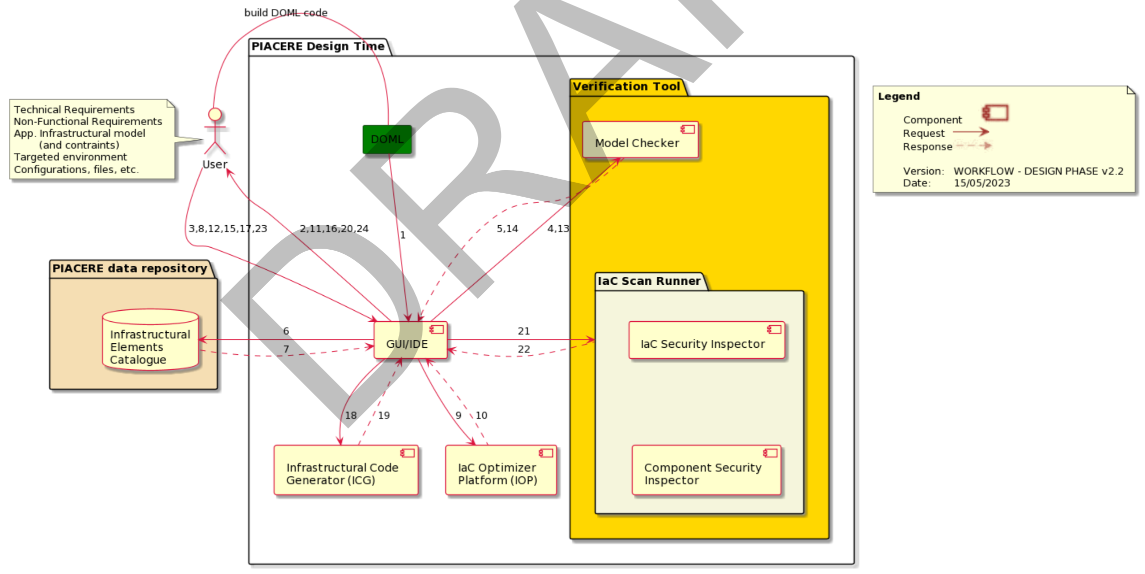

Figure 1. DOML positioning in the PIACERE approach.

Figure 1 shows the positioning of the DOML in the PIACERE ecosystem. The numbers associated to the arrows are described in detail in [8] and represent the steps of the workflow activated when a user wants to create a DOML model. More specifically, the user exploits the IDE to design the model. The DOML ecore [5] and Xtext [6] representations allow the IDE to support the user by providing suggestions about the syntax to use. Through the IDE, the user can exploit all other tools in the figure. More specifically, through the Infrastructural Elements Catalogues he/she can obtain information about the resources that are known to PIACERE. Through the Model Checker it is possible to verify the correctness of the model in terms of internal consistency and fulfilment of explicitly defined requirements. Through the IOP it is possible to obtain a mapping of the abstract infrastructure layer into a concrete infrastructure, through the IGC the generation of IaC and, finally, through the IaC Scan Runner the verification, from the security perspective, of the final code.

The DOML ecore and Xtext implementations are available in the PIACERE public repository together with a tutorial presenting the usage of the DOML[1]. In the following of this section, the main characteristics of the DOML are presented together with the metamodel that regulates its usage and the language syntax to be used by external users. Furthermore, DOML extension mechanisms are explained, and finally, the DOML supporting components are presented. These include: i) a web service that translates DOML specifications in an internal format used by some of the other PIACERE tools, the DOMLX, and vice versa; ii) a component called DOMLIZER that is in charge of receiving resource descriptions available in the catalogue and transforming them into DOML fragments.

2.1.1 DOML main characteristics

The DOML provides the following main characteristics.

- *Reduce the need for polyglotism in DevOps teams*: as it has been mentioned in the previous section, typical IaC approaches are focused on specific aspects of the system life cycle. For instance, Terraform is focused on resource provisioning, Ansible on the configuration and execution of software layers on top of existing resources, container technologies such as Docker on the creation of proper execution environments for software components, orchestrators such as Kubernetes on the management of containers and their possibility to scale in/out. This implies the need to have a polyglot DevOps team able to show a reasonable level of proficiency on multiple IaC languages and approaches. DOML, in combination with the translation features offered by ICG [9], allows its users to define a single DOML model that, at the time of writing, can be used for resource provisioning and software configuration and execution. While the other aspects that have been enumerated are not covered yet, they can be potentially addressed through the development of ICG templates developed to translate the DOML model into proper target IaC languages. This will be the subject of future work beyond the scope of the PIACERE project.
- *Facilitate the instantiation of an abstract infrastructure on top of different resource providers*: one of the issues that is often faced by DevOps teams is the need to deploy a specific application on different cloud providers. This is due to multiple reasons, ranging from the opportunities deriving from special deals offered by different providers in different cases to the need to avoid so-called vendor locking, that is, the case when it becomes very expensive and difficult for the team to move to a different provider.
  The multi-layered approach offered by the DOML has the potential to reduce the effort of the team in exploiting resources from different providers. In fact, the application and the abstract infrastructure layers of a DOML model can be reused when moving from one provider to the other, while associations to multiple providers' resources can coexist in the same model.
- *Keep the external representation of a DOML model separate from the internal one*. As it has been mentioned before, the DOML internal representation is separated from the external one. Thanks to this design choice, different external representations can be associated to the same DOML model. This can be useful to accommodate preferences of different users. While the first external representation developed is a textual one, as part of the IDE development, the usage of Eclipse-based frameworks that support the

[1] https://git.code.tecnalia.com/piacere/public/the-platform/doml

creation of a graphical representation has been experimented. A proof of concept of this is under development and will be made available by the end of PIACERE project.

2.1.2 DOML metamodel

The DOML metamodel consists of several “layers”, which incrementally enrich the description of the cloud-based applications that will be managed inside PIACERE. Each layer provides a unique point of view of the applications; yet all the layers are built up for a comprehensive application description. This approach allows developers to describe how cloud applications are structured in an abstract manner, mapping the different software components to the concrete infrastructure elements, enabling the usage of different concretizations to match one particular deployment.

2.1.2.1 Commons Layer

The **Commons Layer** contains the main abstract application agnostic concepts that are shared among different layers (see Figure 2). The DOML extension mechanisms (DOML-E) are also addressed in this layer by setting up the basic elements that will allow creating new concepts and properties in the top layers.

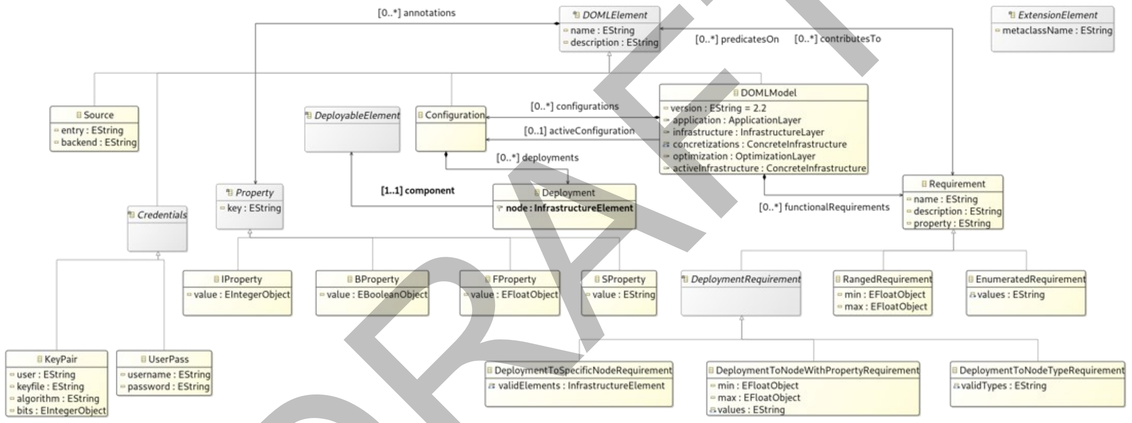

Figure 2. Commons Layer diagram

2.1.2.2 Application Layer

The **Application Layer** (see Figure 3) contains the information to describe the components and building blocks that compose the applications, as well as the functional requirements of each of them in terms of software interfaces and APIs. Finally, this layer describes how the application is deployed into the different infrastructure components.

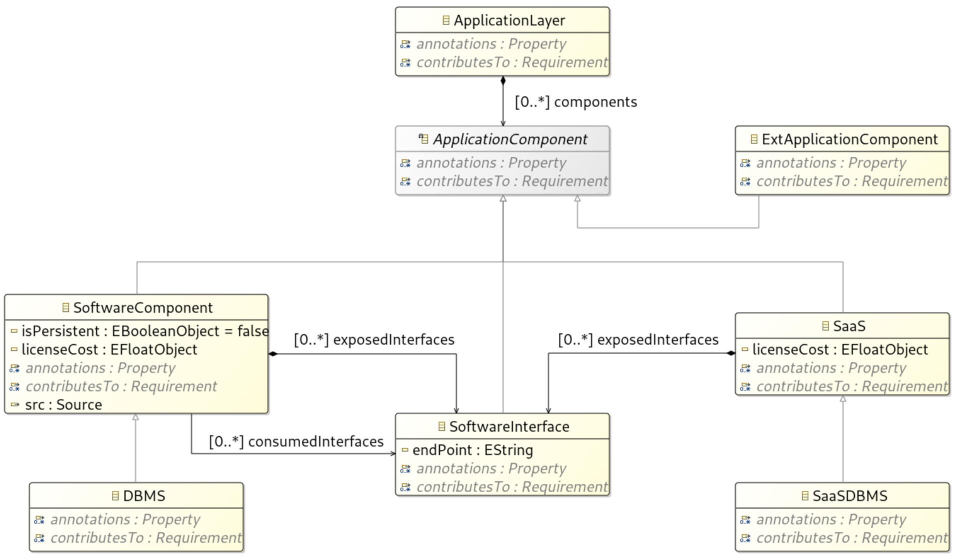

Figure 3. Application Layer diagram

2.1.2.3 Infrastructure Layer

The **Infrastructure Layer** (see Figure 4 for an overview) defines the abstract infrastructure elements that will be used to deploy the application components. The readability of the figure could not be improved for lack of space. A detailed description of all its elements is available in the Annex [2]. Concepts in this layer will include information that is relevant to meet the requirements of the applications. However, most of the concepts in this layer will require a concretization, or in other words, a more concrete instance they will be mapped on. For example, a virtual machine (VM) in this layer must be mapped to a concrete virtual machine instance, either a VM from AWS or a specific VM deployed by the user.

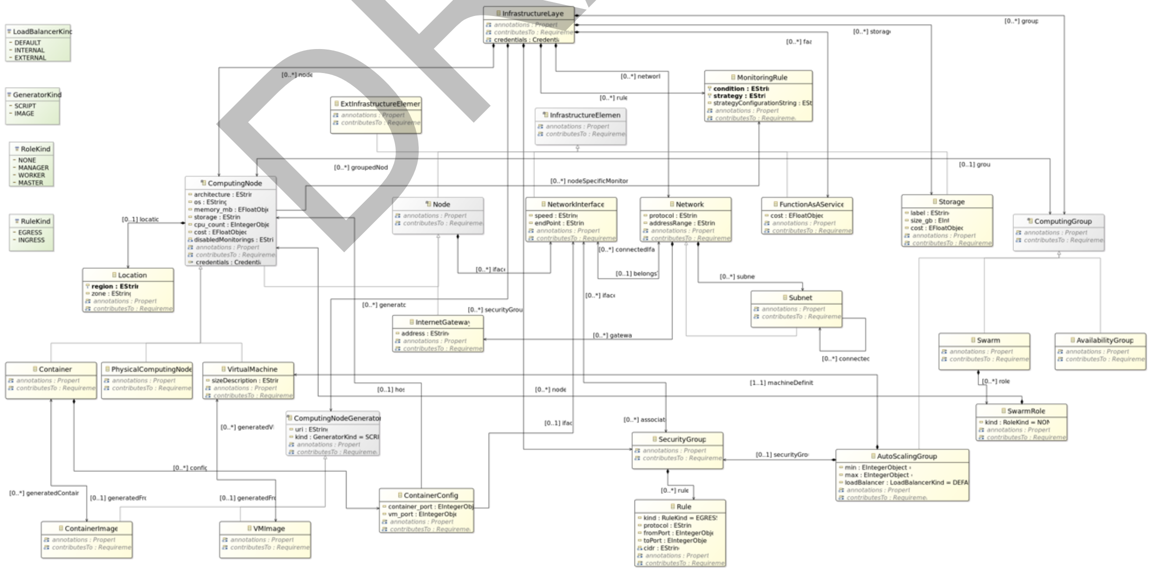

Figure 4. Infrastructure Layer diagram

2.1.2.4 Concrete Layer

The **Concrete Layer** (see Figure 5) provides the tools to concretize the infrastructure elements in the Infrastructure Layer and map them onto specific infrastructure instances either provided by cloud runtime providers, such as AWS or Google Cloud, or provided by the users.

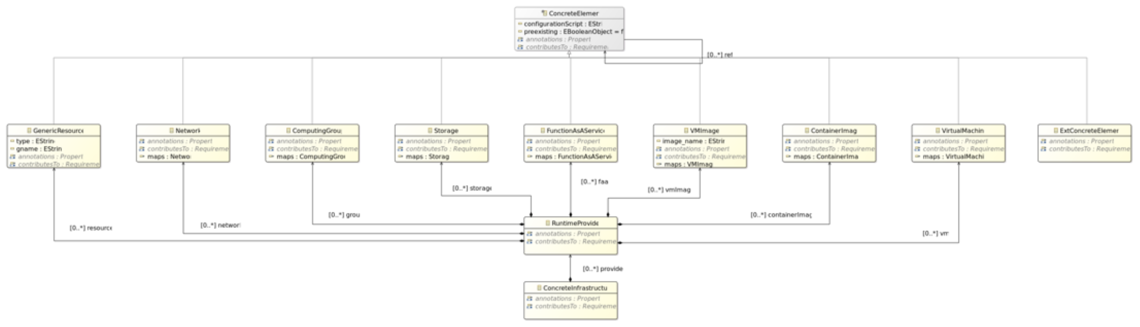

Figure 5. Concrete Layer diagram

2.1.2.5 *Optimization Layer*

The **Optimization Layer** (see Figure 6) defines all the information required for the optimizers to locate the best configurations for cloud applications described with the DOML, such as optimization objectives and non-functional requirements, as well as means to capture the optimization solutions.

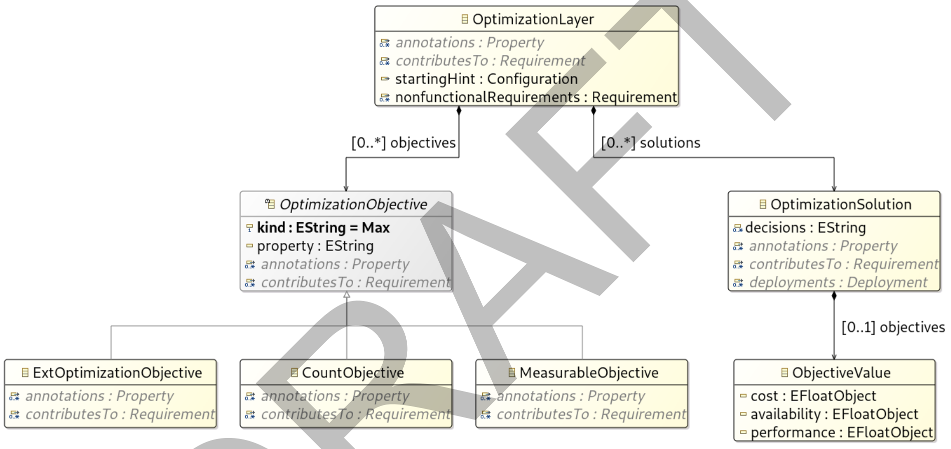

Figure 6. Optimization Layer diagram

2.1.3 DOML syntax

In order to offer to end users a tool to define a DOML model, a syntax has been developed by means of the Eclipse Xtext™ [6] framework. Most approaches make the text syntax directly derive from the metamodel (e.g., using tools such as the wizard provided by Xtext); however, this kind of approaches leads to a one-to-one mapping of each element in the metamodel. To achieve a high degree of flexibility for the language and improve its readability and ease-of-use, it has been decided to use a different approach, developing an ad hoc syntax, yet carefully aligned with the metamodel.

Diagrams showing the fully detailed syntax are available in [2] and a tutorial for beginners to get familiar with the syntax can be found in [3].

The syntax structure follows the layered, incremental approach used for the metamodel. In a DOML model, an application can be described in four layers: application layer, abstract / concrete infrastructure layer and optimization layer. In a declarative manner, layers are described in sequence, allowing the user to completely describe an application from all the different viewpoints.

Most of the elements can be introduced by means of a keyword and a unique identifier: all the aspects related to such elements can be described through the usage of key-value pairs defined within curly braces. A set of attributes is defined for each kind of element, covering all the main features. Some code examples can be found in Section 4.

2.1.4 DOML-E

In order to meet the needs of the continuously evolving cloud markets, DOML includes extension mechanisms that allow the users to create new concepts from the existing ones. These extensions mechanisms are referred to as DOML-E. The DOML is currently extended in the following ways:

- *Creation of new concepts*. The new concepts will require the definition of a metaclassName. Extension elements exist in all the DOML layers, e.g., the Application Layer includes the class ExtApplicationComponent that incorporates into the Application Layer a new type of ApplicationComponent. Further details on the definition of these extension classes are provided in the Annex.
- *Definition of new properties*. The set of properties and attributes associated to one particular DOML concept can be extended to further increase its expressiveness.
- *Usage of the GenericResource concept*.

In the following subsections**,** the detailed steps for extending DOML are illustrated. More details on the implementation-level steps for extending metamodel are presented in Section 2 of the Appendix “Further details”.

2.1.4.1 Creation of New Concepts

Suppose that a new service concept and a new docker service concept would need to be introduced in the infrastructure layer.

To this end, the metamodel will be firstly modified by creating in the Commons package a metaclass named Service which extends the abstract class DOMLElement and includes the necessary attributes and references. For the sake of simplicity and clarity, the following example Service has been created:

```
class Service extends DOMLElement {
      attr Integer port;
      attr String [*] constraints;
}
```

In the above example, a Service is a DOMLElement which has a port attribute and several possible constraints expressed with String (for simplicity).

Now considering that the container service is simultaneously a computing node and a service, multiple inheritance is used to model this concept:

```
class Container extends ComputingNode, commons.Service {
      ref ContainerImage #generatedContainers generatedFrom;
      val ContainerConfig [*] configs;
}
```

In above example, the existing container has been extended by another superclass. Note that the references inside the class are associated to its ComputingNode characteristic.

Now the corresponding concrete syntax is ready to be created for them. For the service, it could be used a grammar fragment, which could be used for any other service-related concept, shown as follows:

```
fragment Service returns commons::Service:
    (
      ('port' port=INT)? &
      ('constraints' '[' constraints+=STRING (',' constraints+=STRING)* ']'
)?
   )
;
```

Since the Container concept can be reused, it has been updated by adding the characteristics as a service:

```
Container returns infra::Container:
   'container' name=ID (('service' '{' DOMLElement Service) | '{' DOMLElement)
      configs+=ContainerConfig*
      ('disabled_monitorings' disabledMonitorings+=STRING
      (','disabledMonitorings+=STRING)*)?
   '}'
;
```

Having introduced the above modifications in the metamodel and in the syntax, the new container service construct can be used. The fragment below is a very simple example of DOML script defined by the above model for a container service:

```
container service dns_server {
   port 53
   constraints [ ‘C1’, ‘C2’ ]
   host vm1 { … }
}
```

2.1.4.2 Definition of New Properties

In current DOML, most concepts contain properties that can be expressed by key-value pairs. This is implemented by adding the property attribute in the superclass DOMLElement, since almost all DOML concepts extend it. The detailed implementation is as follows:

```
abstract class DOMLElement {
   ...
   val Property [*] annotations;
}
abstract class Property {
   attr String key;
   ref DOMLElement reference;
   op Object getValue();
}
class IProperty extends Property {
   attr Integer value;
}
```

The above fragment of DOML metamodel shows an example of integer property definition. Other properties like string, float, etc. are defined analogously.

A doml script example is as follows:

```
faas concrete_f {
   properties {
      lambda_role_name = "DemoLambdaRole"
      lambda_runtime = "python3.8"
      lambda_handler = "image_resize.lambda_handler"
      lambda_timeout = 5
      lambda_memory = 128
   }
   maps f
}
```

In the above example, different types of properties are defined for a concrete FaaS component of an application.

When defining new properties, the Infrastructural Code Generator should be updated accordingly. In the ***User Manual in Deliverable D3.6*** [9] an explanation on how to do it can be found.

2.1.4.3 Usage of the GenericResource concept

The GenericResource class has been introduced in DOML 2.2 to support some concrete generic resources that are specialized in different IaC languages.

For any generic resource, the current version of DOML allows to specify its type and name.

The DOML definition of a generic resource is as follows:

```
generic_resource vsphere_dc {

   type "datacenter"
   gname "dc1"

}
```

This class is a specialization of the ConcreteElement class and, as such, it inherits the "preexisting" attribute and the "refs_to" association to another ConcreteElement object: this is particularly relevant, since this feature could be used to model already existing infrastructural resources.

```
generic_resource vsphere_cl {

   preexisting true
   refs_to vsphere_dc
   type "compute_cluster"
   gname "cl1"

}
```

Examples of what generic resources could model are the following: data centers, clusters, pools, etc.

2.1.5 DOML supporting components

2.1.5.1 DOMLIZER

The objective of the DOMLIZER is to simplify the access of new DOML users to existing resources, like, for example, existing virtual machine descriptions, resources made available by runtime providers, etc. The PIACERE framework includes an element called the Infrastructure Elements Catalogue-IEC (hereinafter the Catalogue), which stores definitions of such existing resources. The information stored in the Catalogue, however, is not in DOML format, as it includes specific data that is relevant for some of the tools in the PIACERE framework, but not to the user (e.g., monitoring and IOP).

The DOMLIZER tool provides the required API to link the elements in the Catalogue to DOML models, performing a translation and including only the information relevant to the user in the model. During the last year DOMLIZER has been improved to provide conversion support for the new types of resources that have been introduced into the Catalogue:

- Existing resource element. This element describes existing virtual machines, databases and storages.
- Image element. This element describes existing virtual machine images.

The DOMLIZER API allows the tools to request a Catalogue element that is returned according to the DOML syntax. Among the other tools, the IDE has been equipped with a shortcut to the DOMLIZER API to easily include Catalogue elements into the current DOML model with just one click.

2.1.5.2 DOML-DOMLX conversion service

The IDE has integrated in the last year a REST service to convert DOML models from the textual DOML files to the XML-based DOMLX format. This tool supports the rest of the IDE tools to easily access the DOML format that best suits them to implement the functionality they are providing.

The implementation of the tool has been done using Xtext and Eclipse Modelling Framework. At the moment, the development of the backward conversion from DOMLX to DOML is under development. To ensure that files converted back from DOMLX are maintainable and readable by users, the service is being implemented using a DOML specific formatter that keeps the DOML files tidy and clean.

The service has been deployed in: https://d2x.ci.piacere.digital.tecnalia.dev/ and it uses the text of the DOML/DOMLX file as input for the conversion.

2.2 Changes in the latest version

Compared to DOML v2.1, some modifications have been applied to both the metamodel and the syntax in DOML v3.0.

Major changes consist in the introduction of some new concepts in DOML. A few concepts (e.g., “DeployableElement”, “Node”) have been introduced to improve the formal representation of some components, while some other concepts have been added to express new concepts needed for the use case scenarios (e.g., “Source”) and to extend the variety of aspects covered in DOML in a proper manner (e.g., “MonitoringRule”, “GenericResource”).
This has led to achieve a higher degree of completeness and expressiveness of the language. Furthermore, some minor changes have been applied to fix inconsistencies between the metamodel and the syntax and to solve some technical issues arisen during the development of use cases.

Finally, a feasibility study has been conducted to improve the uniformity and conciseness of the syntax, which was consequently slightly modified to be easier to use.

Further details on such changes can be found in the Annex [2].

2.3 Functional description and requirements coverage

The development of the DOML has been guided by the requirements that have been defined with the collaboration of all PIACERE partners as part of Deliverables D2.1 [10] and D2.2_v1.1 [8] (this last one provides the latest version of such requirements and of the PIACERE architecture). Additionally, the work done has been based on the definition of specific scenarios that have guided the development of the new version of DOML from the beginning of the second year of the project. In this section, it has been provided a summary of the current accomplishment of the requirements relevant to the DOML and a final version of the specific scenarios fulfilled by the language.

2.3.1 Requirements coverage

For the sake of clarity, the requirements are split in two tables, one focusing on the general characteristics of the DOML (Table 2) and another concerning the elements of applications and infrastructures the DOML should represent (Table 3). For each requirement, an explanation of the level of achievement is provided together with an explanation. Requirements have been also reordered to have the most general ones at the beginning of the Tables 2-3 followed by more specific ones. For the sake of traceability, the requirement identifiers defined as part of WP2 have been kept.

Table 1. Requirements on the general characteristics of DOML.

Req ID	Description	Level of achievement and justification
REQ63	DOML must be unambiguous.	**Achieved**: DOML is formally defined in terms of its translation into the corresponding IaC code fragments. As such, it is not ambiguous by definition.
REQ62	DOML must support different views.	**Achieved**: DOML allows models to be defined on a per-layer basis. Layers represent different viewpoints on the system: 1) in the application layer, the definition of the application components and the dependencies between them. 2) in the abstract infrastructure layer, an abstract definition of the needed infrastructure, represented in terms of categories of elements and their mapping with the application-level components they are in charge of executing. 3) in the concrete infrastructure layer, a definition of the proper configuration information for the concrete infrastructure elements to be used and their association to the corresponding abstract elements. 4) In the optimization layer, a definition of multi-objective optimization problem of infrastructure resource provisioning.
REQ70	The DOML should allow users to state correctness properties in a suitable sub-language (possibly Formal Logic).	**Achieved**: This requirement is addressed in the DOML in two different ways: 1) The main correctness relationships among elements in the specification are directly defined as part of the language semantics and are verified by the Model Checker [11]. 2) It is also possible to express in the DOML some generic constraints that, once again, are verified by the Model Checker.
REQ76	DOML should allow the user to model	**Achieved**: DOML models include the information relevant to the listed phases.

	information needed for each of the four considered DevOps activities (Provisioning, Configuration, Deployment, Orchestration)	
REQ57	It is desirable to enable both forward and backward translations from DOML to IaC and vice versa	**Partially achieved**: DOML currently supports to the forward translation to different IaC, e.g., Terraform and Ansible. Enabling backward translations could open up the possibility to incorporate existing IaC definitions into the DOML, thus increasing reuse and the potential impact of the DOML itself. For the above reason, we considered this as an interesting requirement. However, it could not be addressed in the timeframe of the project, given the complexity of the forward translation that had to be studied and considered in several different cases.

Table 2. Requirements on the specific elements to be modelled in DOML.

Req ID	Description	Level of achievement and justification
REQ01	The DOML must be able to model infrastructural elements.	**Achieved:** This requirement is addressed by the DOML by offering primitives to represent the most relevant infrastructural elements: containers, virtual machines, network elements, security groups, etc. Clearly, the exhaustive definition of infrastructural elements as base types in the DOML is not possible. For this, the DOML will offer the possibility to define new elements through the extension mechanisms (DOML-E).
REQ25	DOML should support the modelling of security rules (e.g., by type tcp/udp..., and ingress/egress port definition)	**Achieved:** This requirement is fulfilled by the new concept of security group, which contains both ingress and egress security rules. It is also possible to specify the communication protocol.
REQ26	DOML should support the modelling of security groups (containers for security rules)	**Achieved**: This requirement is addressed by a specific construct in the language.
REQ27	DOML should support the modelling, provisioning, configuration, and usage of container engine execution technologies (e.g., docker-host)	**Achieved**: The DOML addresses this requirement by offering constructs to define a container, a container image, and a container file. A container can then be mapped on multiple hosts and ports.
REQ28	DOML should support the modelling of containerized application deployment (e.g., pull/run/restart/stop docker containers)	**Achieved**: As stated for REQ27, the DOML offers the possibility to model containers and its constituents. As stated for REQ76, the DOML does not support the explicit modelling of workflows to which the pull/run/restart/stop activities belong to. DOML, however, supports the possibility to link elements to script files of various kinds. This opens up the possibility to define specific low-level operations on containers as part of these script files.

REQ29	DOML should support the modelling of VM provisioning for different platforms such as (OpenStack, AWS) for canary and production environments	**Achieved**: This requirement is fulfilled with the possibility to support different platforms for VM provisioning.
REQ30	DOML should enable support for policy definition constraints for QoS/NFR requirements	**Achieved**: DOML supports the definition on QoS/NFR requirements (see REQ61) as well as the definition of monitoring rules. Further needs for the definition of additional policies did not emerge so far.
REQ58	DOML should offer the modelling abstractions to define the outcomes of the IoP	**Achieved**: This requirement has been fulfilled by introducing the optimization solution concept which is composed of the results of the optimization together with the decision variables.
REQ59	The DOML should allow users to define rules and constraints for redeployment, reconfiguration, and other mitigation actions	**Achieved**: The DOML addresses it by supporting the definition of the requirements and constraints that should be considered while performing mitigation actions. These concern, for instance: • the structural characteristics of the infrastructural elements to be used (if the user states that a VM with 16 GB of RAM should be used for executing a certain application component, any change of VM should ensure that this requirement is still fulfilled) or • the definition of non-functional requirements predicating on response time, availability, or other characteristics of application components. Moreover, the DOML language supports the definition of monitoring rules, in terms of monitoring conditions that can trigger the execution of other monitoring and reconfiguration strategies and the configuration of such strategies.
REQ60	DOML should support the modelling of security metrics both at the level of infrastructure and application	**Achieved**: DOML now includes a specific syntax to describe security metrics and monitoring rules associated to them. Now the DOML supports the definition of strategies based on Ansible playbook to enforce some security strategies.
REQ61	DOML must support the modelling of NFRs and of SLOs	**Achieved**: NFRs and SLOs definition is supported in DOML and used to describe the constraints for the IOP (infrastructure optimization).
REQ36	DOML to enable writing infrastructure tests.	**Partially achieved**: Infrastructure testing typically focus on injecting faults in specific points of the infrastructure and then observing the reaction of the system. Chaos engineering is the discipline that focuses on this aspect. A study on the tools adopted in chaos engineering has been conducted as part of a thesis work [12]. Some of the available tools can be configured with the definition of the infrastructure and application to be tested. In this respect, a DOML model describing an application and the underlying infrastructure can potentially be used as an input for such tools. This aspect will be explored as future work beyond the end of the project.

REQ111	The user could have the possibility to add external custom own IaC	**Achieved:** This requirement has been introduced in the last project year to enable reuse of pre-existing IaC artifacts. The DOML has been modified to allow the user to specify that a software component can be managed through a pre-existing source file interpreted by an engine that can be specified as part of a DOML model. More details on this point are described in Section 2.3.2.4.

2.3.2 Scenarios

This section presents the scenarios of DOML usage that have been specifically analysed and addressed starting from the second project year. The structure of these scenarios is aligned with the guidelines associated to agile development using Gherkin syntax (https://cucumber.io/docs/gherkin/). They concern the following aspects:

- Create an empty DOML model and insert elements in a guided way in the model.
- Define a container and associate a software component to it.
- Associate a software component to specific IaC code.
- Create an autoscaling group.
- Define functional and non-functional requirements.
- Extend the DOML with new resources/providers.

The main feature to focus on is the following.

```
Feature:  Creation of a new DOML for a specific software application

   As a PIACERE user I want to create a new DOML model to automate the
   provisioning of the corresponding resources and the deployment of the
   whole software stack and its configuration
```

The following scenarios defined have been all realized. While the purpose and the main characteristic of each scenario remained the same through the development of the DOML, the adopted DOML syntax has been improved. The one shown in the following subsections is the last one aligned with DOML 3.0.

2.3.2.1 Scenario 1: create an empty DOML model

```
Scenario: Create a new empty DOML model
Given An installed PIACERE IDE
When user starts a new PIACERE DevOps project
Then a new DOML file is created
```

This scenario has been implemented as-is.

2.3.2.2 Scenario 2: insert a new DOML element in a model with the guidance of the IDE helpers

```
Scenario: Insert a new DOML element in a DOML model
Given An empty DOML model
When user starts typing the keyword software_component or infrastructure or
...
And continues with an identifier for the element to be added
And adds needed details (properties or attributes defined for the specific
element type)
Then the new element is created
```

This scenario has been implemented as-is.

2.3.2.3 Scenario 3 and 4: define a container and associate a component to it

```
Scenario: Define the container as part of the infrastructure and associate a
component to it
Given a DOML model
When user digits something like:

software_component c1 {
  source s1 {
    // e.g., ansible_code.yml
    entry "..."
    // e.g., ansible
    backend "..."
  }
}
...

container co1 {
  host vm1 {
    ...
  }
}

cont_image co1_img {
  generates co1
  image "docker.hub.io/myhub/co1:1.0"
}

vm vm1 {
  os "CentOS-7-2111"
  cpu_count 2
  mem_mb 8192.0
  iface i1 {
    //belongs_to net1
  }
}

...

deployment infra_config {
  c1 -> co1
}


Then software_component c1 is meant to be deployed within the container co1,
which is created from the specified image and is mapped into vm vm1.
```

In D3.2 [1] this scenario was divided in two different ones. Here they have been grouped together as the container details are not anymore defined within the definition of a component as it was in the initial idea. In DOML 3.0 the container instantiation is handled through the usage of a specific syntax to describe from which image has been generated. The association between a container and a VM is managed as a host configuration within the container definition, instead of being specified in the deployment section. The credentials-related part is not necessary anymore, since the access to the docker repository, where the container image resides, is managed by the PIACERE IEM component [13].

2.3.2.4 Scenario 5: associate a software component to a specific IaC code

```
Scenario: Associate a software component to specific IaC code
Given a DOML model
When a user digits something like the following
software_component nio3_git {
  source s1 {
    entry "ansible/provision.yml"
    backend "ansible"
  }
  properties {
    nexus_docker_registry_user = "***";
    nexus_docker_registry_password = "***";
  }
}
The software_component nio3_git relevant code is found in a folder in the
local workspace. The code is meant to be executed starting from the specified
entry, with the specified backend (Ansible in the example).
```

Compared to its previous version in D3.2 [1], this scenario has been slightly changed, since it has been decided to have the relevant code for the necessary software components located in a folder in the project workspace instead of supporting the usage of external URIs in DOML v3.0.

2.3.2.5 Scenario 6: create an autoscaling group

```
Scenario: Create an autoscaling group
Given a DOML model
When a user digits:
autoscale_group ag {
  vm vm_template {
    cpu_count 2
    mem_mb 1024.0
    iface i1 {
      belongs_to net1
    }
    credentials ssh_pass
  }
  min 1 max 2
}
Then autoscaling group is created,
And it contains a template for creating a VM instance with the specific
requirements on CPU, memory, etc.
And the scale is specified by the minimum and maximum number of VMs
```

This scenario hasn't changed with regards to the previous version presented in D3.2 [1], since there were not modifications related to autoscaling groups in the DOML syntax from v2.1 to v3.0.

2.3.2.6 Scenario 7: define functional and non-functional requirements

```
Scenario: Define functional and non-functional requirements
Given a DOML model
When a user digits:
functional_requirements {
  req_ext ```
  >   "example requirement to test"
      # Expr to parse
      not (
        vm is class infrastructure.VirtualMachine
        and
        vm is not class infrastructure.Storage
        or
        vm is not class infrastructure.Storage
        implies
        vm is class infrastructure.Storage
      )
      iff
      not exists iface, apple (
        forall orange (
          vm has association infrastructure.ComputingNode->ifaces iface
          or
          vm has association infrastructure.ComputingNode->ifaces iface
```

~~~
        )
        and
        vm has attribute infrastructure.ComputingNode->os Os1
      )
      ---
      "Virtual Machine {vm} has no iface"
  ```;
}
Then functional requirements are created (which can be some external
requirements in external DSL like the example)
And they are dedicated to the verification tools
When a user digits:
nonfunctional_requirements {
  req1 "Cost <= 70.0" max 70.0 => "cost";
  req2 "Availability >= 66.5%" min 66.5 => "availability";
}
Then the nonfunctional requirements are created (which can be some numerical
constraints like the example)
And they are dedicated to the Optimization tools
~~~

This scenario has been implemented as-is.

2.3.2.7 Scenario 8: extend the DOML with new resources

```
Scenario: Extending the DOML with new resources
Given the existing DOML metamodel
When a user creates a new class extending a specific DOML metaclass for a new
resource
And adding the needed attributes and references to other elements in the class
And creating the desired concrete syntax in the grammar definition
Then a new DOML supporting a specific new resource is created
```

This scenario has been implemented as-is, as discussed in Section 2.1.4.

2.4 Main innovations

As discussed in Section 2.1.1, one of the main innovations introduced by the DOML concerns the possibility to define an infrastructure at an abstract level, using provider and technology-independent concepts (e.g., virtual machine, container, network, …), and to instantiate it in multiple concrete infrastructures. Such concrete infrastructure definitions can coexist in the same DOML model. Moreover, the selection of the concrete infrastructures to adopt can be delegated to the IOP component that will select the optimal one or more based on criteria that are defined as part of the DOML model itself, thus reducing the effort required to DevOps teams.

To evaluate the effectiveness of the DOML from the simplicity-to-use perspective, in [7], DOML models have been compared with Terraform and TOSCA/Cloudify-based specifications. The analysis highlights that Terraform and TOSCA/Cloudify IaC is heavily dependent on the selected provider, where each provider brings in a custom set of resources showing different parameters and configuration possibilities that must be mastered by the user willing to create an infrastructure. Another issue is the difficulty of acquiring from a Terraform code fragment an overview of the entire system to be run as the attention is exclusively focused on the infrastructural aspects.

The possibility offered by DOML to define in a single model both the application and the underlying abstract and concrete infrastructure opens up to the possibility to verify the correctness of the model and, therefore, the instantiation of the application on top of the infrastructure and, in turn, the mapping of the abstract infrastructure on a concrete one. This verification is achieved through the model checker that is integrated within the PIACERE IDE and fully compatible with the DOML.

Another important innovation concerns the clear distinction that has been introduced between the external DOML representation and the corresponding internal one (also called DOMLX). The DOML-DOMLX conversion service (see Section 2.1.5) is the new component in charge of managing the DOMLX and any needed translation. Thanks to its availability, the compatibility between different DOML versions can be managed by translating an external DOML representation written in one version of the language into DOMLX and then, from this, back into a different version of the external language representation.

Finally, other innovations critical to the consolidation of the whole PIACERE project consist in the following aspects:

- Consolidation and clean-up of the DOML modelling language which has now reached version 3.0. Through the PIACERE project, the following main versions of the language have been released: 1.0 at the end of the first project year, 2.0 and 2.1 during the second project year, 2.2.1 and 2.2.2 in the third year.
- Consolidation of the extension mechanisms (DOML-E).
- Transformation of the resource descriptions in the PIACERE catalogue into DOML fragments to support users in selecting resources and incorporating them in a DOML model. This is done through the supporting component called DOMLIZER.
- Elaboration of the examples of use of the DOML language using the PIACERE IDE, their validation through the Model Checker, their optimization through the IOP tool, and the generation of the corresponding IaC code through the ICG.

3 Overview of preliminary experiments

The DOML is currently being used and experimented in the PIACERE case studies and it is expected to have a complete evaluation, which will include also DOML-E, at the end of the project.

Until now, the following actions have been performed:

- defined multiple examples that can be called reference test cases and that can be used to check whether the DOML fulfils the expectation and whether the other tools of the PIACERE toolchain work as expected;
- conducted a first rigorous empirical evaluation by comparing the performance of the DOML with the one of two well-known existing IaC languages, that is, Terraform and Cloudify.

In Section 3.1 , a short summary of the empirical evaluation is presented while in Section 3.2 the reference test case examples are discussed.

3.1 Empirical evaluation

In [7] it has been conducted a first evaluation of the DOML by addressing two specific research questions:

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

The first research question has been addressed by comparing the IaC code defined by independent researchers in [14] using two different languages, Terraform and Cloudify with an equivalent DOML model. The objective results obtained in this experiment are summarized in Table 3 where, for each of the three considered approaches, the number of lines of code (#LOC), the number of files (#File) composing the specifications, and the number of used languages (#Languages) are listed.

Table 3. Comparison between DOML and other IaC approaches.

Approach	#LOC	#Files	#Languages
DOML	103	1	1
Terraform	305	3	2
Cloudify	506	9	2

Essentially, the table shows that the DOML model taken as example is more concise compared to its counterparts as it requires less lines of code and does not require the usage of additional external languages that are instead required by Terraform and Cloudify[2].

[2] The reader should note that, as described in Section 2.3.2.4, a DOML model can be linked to external scripts. Such possibility allows users to integrate legacy IaC into a DOML model, but this is not mandatory for supporting the provisioning and configuration operations.

As for the second research question, it has been shown that a DOML model can target multiple providers by specifying their resources within different concretizations but keeping the abstract infrastructure layer untouched.

3.2 Reference test cases example

3.2.1 Introduction and comparative table

During the development of the DOML, four "test case examples" have been developed aiming at exercising the specific elements of the language in different contexts. These examples are the following:

- a web application using NginX as web server and a DBMS;
- a WordPress application;
- an application exploiting the FaaS (Function as a Service) paradigm;
- a simplified version of the case study developed by Ericsson for PIACERE and focusing on the configuration of a relatively complex network infrastructure.

The NginX and the Wordpress applications are presented in this section while, for the sake of brevity, the Ericsson case is available in the Appendix (Section 7) and the FaaS example, which resulted not to be of interest of the PIACERE partners, is available on the DOML public repository[3]. The selection of these specific examples aims at ensuring a significant coverage of the language elements, as summarized in Table 4, where, for each DOML layer, the concepts used by each example are listed.

Example	Application Layer	Infrastructure Layer	Concrete Layer	Optimization Layer
NginX	• DBMS • Software component	• VM • Network with Internet gateway • Interconnected subnets • Key pair credentials • Security group	• Concrete VM • Concrete Network	• 2 Optimization Objectives • NFR: Region requirement • NFR: Provider requirement • NFR: Elements to • be deployed requirement
WordPress	• DBMS • Software component • Alternative example with SaaS DBMS	• VM • Network with subnet • Autoscaling group • Key pair credentials • Security group • VM image • Container	• Concrete VM • Concrete Network • Concrete Autoscaling group • Concrete pre-existing VM image with a given image name	/

[3] https://git.code.tecnalia.com/piacere/public/the-platform/doml

		• Container image	• Concrete pre-existing container image • Example of two different concrete infrastructures using different providers	
FaaS	• Software component • Source code for a given software component • SaaS	• Network • Container • Container image • Key pair credentials • FaaS • Storage • Autoscaling group • Security group	• Concrete Network • Concrete Autoscaling group • Concrete FaaS • Concrete Storage • Concrete pre-existing VM image • Concrete pre-existing container image	/
Simplified Ericsson	• Software component • SaaS • Example of provided software interface at a given URL	• VM with more than one network interface • Several networks with different interconnected subnets • Networks with Internet gateways • Key pair credentials • Security group	• Concrete VM • Concrete Network	• 3 Optimization Objectives (all the available ones) • NFR: Cost requirement • NFR: Performance requirement • NFR: Provider requirement • NFR: Elements to be deployed requirement

Table 4. Comparative table for test cases examples

3.2.2 WordPress Website

WordPress is a popular open-source Content Management System (CMS) that can be used to easily develop blogs and other kinds of websites. WordPress is written in PHP programming language, so it needs to be run on a server with the appropriate runtime environment properly configured. It also needs a SQL database as a backend for storing website data.

The structure of the WordPress application is that: a WordPress is running in a container which is hosted in a VM provisioned by a provider, e.g., AWS. VM is defined in an autoscaling group where the size is defined as two. WordPress is connecting to a database through network.

The code of the example is reported below, divided into several code snippets, which are explained separately. In practice, all such code snippets are merged in a single DOML file.

```
doml wordpress
```

In the first line of the file, the name of the DOML model is declared.

3.2.2.1 *Application Layer*

Next, the application layer is defined.

```
application app {
    dbms postgres {
         properties {
             identifier = "education"
             name = "wp_db"
        }
        provides {
            SQL_interface
        }
    }

    software_component wordPressServer {
        consumes {
            SQL_interface
        }
    }
}
```

The application layer consists of two components: the postgres database and the WordPress server. Here the relationships between these components are declared: the database provides a SQL interface, which is consumed by the WordPress server. Additionally, here the properties of the DBMS are defined. They are agnostic with respect to the DBMS implementation in the infrastructure layer, examples are its name and identifier. The database credentials are not handled directly in the DOML code, since having them unencrypted would result in a bad practice, leading to potential security issues. Instead, they are managed by other components in the PIACERE framework.

3.2.2.2 *Abstract Infrastructure Layer*

The abstract infrastructure layer is shown below:

```
infrastructure infra {
    net net1 {
        cidr "10.0.1.0/24"
        protocol "TCP/IP"
        subnet subnet1 {
           cidr "10.0.1.0/24"
        }
    }
    vm dbms_vm {
        os "ubuntu"
        size "micro"
        iface dbms_iface {
          belongs_to subnet1
          address "10.0.1.2"
        }
        credentials ssh_key
    }

    autoscale_group ag {
        vm wp_vm {
           os "ubuntu"
           size "micro"
           cpu_count 2
           mem_mb 1024.0
           iface i1 {
              belongs_to subnet1
              address "10.0.1.1"
           }
           credentials ssh_key
        }
        // count = 2
        min 2 max 2
    }

    vm_image vm_img {
        generates wp_vm
    }

    key_pair ssh_key {
        user "user"
        keyfile "ssh key"
    }

    container container1 {
        properties {
           WP_DB_HOST = "dbms_vm"
           WP_DB_USER = "username"
           WP_DB_PASSWORD = "password"
           WP_DB_NAME = "database.name"
        }
        host wp_vm {
           container_port 80
           vm_port 8080
           iface i1
        }
    }

    cont_image container_image {
        generates container1
        image "docker.hub.io/wordpress/wordpress:5.8.0"
    }

    security_group sg {
        egress icmp {
           from_port -1
           to_port -1
           protocol "ICMP"
           cidr ["0.0.0.0/0"]
        }
        ingress http {
           from_port 80
           to_port 80
           protocol "TCP"
           cidr ["0.0.0.0/0"]
        }
        ingress https {
           from_port 443
           to_port 443
           protocol "TCP"
           cidr ["0.0.0.0/0"]
        }
        ingress ssh {
           from_port 22
           to_port 22
           protocol "TCP"
           cidr ["0.0.0.0/0"]
        }
        ifaces i1, dbms_iface
    }
}
```

First, the network is defined, with the specification of the CIDR and protocol and the definition of a subnet. The database VM is defined by specifying a set of attributes, which will determine the VM characteristics. The autoscaling group is defined by providing the template of virtual machine “wp_vm” and the minimum and maximum number of VMs supported. A VM image is used to generate such virtual machine. The key-pair credentials for the virtual machines are then defined. In the next fragment, “container1” is to be hosted on the virtual machine “wp_vm” with the port mapping “8080:80” binding with network interface “i1”. The container is generated from a Docker container image, as specified in the following code fragment. Finally, the security group defines several egress rules (w.r.t. ICMP) and ingress rules (w.r.t. HTTP, HTTPS and SSH) for the network.

Note that no VM image for generating the database VM is provided explicitly. Thus, the ICG will provide one automatically during deployment, by inferring its requirements from the properties of the database component at the application layer, and its being linked with “dbms_vm” and the concrete VM mapping it in the concrete layer.

Each component from the application layer is linked to the abstract-infrastructure component that implements it in the following deployment configuration:

```
deployment config1 {
    wordPressServer => container1,
    postgres => dbms_vm
}
active deployment config1
```

The last line states that the deployment configuration above is currently active. In principle, multiple deployment configurations could be defined and switched.

3.2.2.3 Concrete Infrastructure Layer

The abstract infrastructure can be concretized in two ways: con_infra1 uses AWS as a cloud provider, while con_infra2 uses OpenStack. The model fragment is shown below:

```
concretizations {
    concrete_infrastructure con_infra1 {
        provider aws {
            vm_image concrete_vm_img {
                preexisting true
                image_name "ami-012e16cfb2f9e8b0a"
                maps vm_img
            }

            autoscale_group concrete_ag {
                maps ag
            }

            vm concrete_dbms_vm {
                maps dbms_vm
            }

            net concrete_net {
                maps net1
            }
            cont_image concrete_wp_image {
               preexisting true
               maps container_image
            }
        }
    }

   concrete_infrastructure con_infra2 {
        provider openstack {
            vm_image concrete_vm_img {
                preexisting true
               image_name "mantic-20230508"
               maps vm_img
            }

            autoscale_group concrete_ag {
                maps ag
            }

            vm concrete_dbms_vm {
                maps dbms_vm
            }

            net concrete_net {
                maps net1
            }

            cont_image concrete_wp_image {
               preexisting true
               maps container_image
            }
        }
     }

     active con_infra1
}
```

In both concretizations, a concrete VM for the “dbms_vm” abstract VM and a concrete autoscaling group for the “ag” abstract autoscaling group are defined and linked through the “maps” statement. Images and network need to be concretized, too. Please note that the “wp_vm” VM must not be concretized, since it is defined as a template for the autoscaling group. The VM image name, which is provider specific, is specified in the concrete layer. Any other relevant provider specific property could also be specified here through the usage of properties. Since the VM image is selected from the provider catalogue and the container image is an existing Docker image, they are both set as preexisting.

Finally, the “active” statement sets con_infra1 as the concrete infrastructure to be used for deployment.

3.2.2.4 Alternative version with SaaS Database

An alternative second version of the WordPress example uses a SaaS database instead of a custom VM deployment. We describe it by reporting only differences compared to the version presented in this section.

The application layer changes by replacing the “dbms” component with the following:

```
saas_dbms postgres {
    properties {
        identifier = "education"
    }
    provides {
        SQL_interface
    }
}
```

Thus, the only difference is that its component type is now “saas_dbms”.

With respect to other layers, the only difference in the Infrastructure Layer is the absence of the “dbms_vm” virtual machine and of its binding with the “database” application in the “deployment” section, and, consequently, the removal of its corresponding “concrete_dbms_vm” concrete instance from both the two concrete infrastructures described in the Concrete Layer.

3.2.3 NginX

This example was developed ad hoc to provide non-expert users with a simple, readable example, showing the syntax and the basic elements to build a DOML model.

Here a web application, which accesses a database is described. Such an application consists of an NginX web server and a MySQL DBMS, which both run on virtual machines provided by the OpenStack cloud provider. Such virtual machines, which should be both located in Europe, are connected to a common network, which also guarantees access to Internet, and each of them has an associated subnet. Finally, these two subnets are connected to each other. Moreover, the availability and performance of the infrastructure must be maximized, and OpenStack must be used as a cloud provider for the infrastructure.

3.2.3.1 Application Layer

The Application Layer is shown below:

```
doml doml_example1

application app_example1 {
   dbms mysql {
      provides {
         sql_interface
      }
   }
   software_component nginx {
      consumes {
         sql_interface
      }
   }
}
```

The Application Layer consists of two components: the NginX web server and the MySQL DBMS. The latter one provides a SQL interface to handle data, which is consumed by the web server.

3.2.3.2 *Abstract Infrastructure Layer*

The Infrastructure Layer is shown below:

```
infrastructure infra_example1 {
   vm nginx_vm {
      arch "x86-64"
      os "Ubuntu-22.04.2-LTS"
      mem_mb 1024.0
      sto "16"
      cpu_count 1
      size "small"
      loc {
         region "00EU"
      }
      iface nginx_iface {
         belongs_to nginx_subnet
      }
      credentials nginx_vm_credentials
   }
   vm mysql_vm {
      arch "x86-64"
      os "Ubuntu-22.04.2-LTS"
      mem_mb 1024.0
      sto "16"
      cpu_count 1
      size "small"
      loc {
         region "00EU"
      }
      iface mysql_iface {
         belongs_to mysql_subnet
      }
      credentials mysql_vm_credentials
   }
   net common_network {
      subnet nginx_subnet {
         connections {
            mysql_subnet
         }
         cidr "10.0.144.0/25"
      }
      subnet mysql_subnet {
         connections {
            nginx_subnet
         }
         cidr "10.0.144.128/25"
      }
      protocol "TCP/IP"
      cidr "10.0.144.0/24"
      gateway igw1 {
         address "10.0.144.22"
      }
   }
   key_pair nginx_vm_credentials {
      user "nginx_user"
      keyfile "ssh key"
      algorithm "RSA"
      bits 4096
   }
   key_pair mysql_vm_credentials {
      user "mysql_user"
      keyfile "ssh key"
      algorithm "RSA"
      bits 4096
   }
   security_group sec_group {
      egress icmp {
         protocol "ICMP"
         from_port -1
         to_port -1
         cidr ["0.0.0.0/0"]
      }
      ingress ssh {
         protocol "TCP"
         from_port 22
         to_port 22
         cidr ["0.0.0.0/0"]
      }
      ingress http {
         protocol "TCP"
         from_port 80
         to_port 80
         cidr ["0.0.0.0/0"]
      }
      ingress https {
         protocol "TCP"
         from_port 443
         to_port 443
         cidr ["0.0.0.0/0"]
      }
      ifaces nginx_iface, mysql_iface
   }
}
```

In this layer, the two VMs hosting the web server and the DBMS are described together with the common network to which they need to be connected to.

The VMs configuration is set through attributes. For each VM, a network interface belonging to the corresponding subnet is defined together with the credentials to be used to access the VMs. The network is configured to have two different subnets connected to each other and a gateway to guarantee Internet access. Finally, a security group is defined to secure access to the network.

Next, the deployment configuration for our application is defined and set as active.

```
deployment config_example1 {
   mysql => mysql_vm,
   nginx => nginx_vm
}
active deployment config_example1
```

3.2.3.3 Concrete Infrastructure Layer

The Concrete Layer is shown below:

```
concretizations {
   concrete_infrastructure concrete_example1 {
      provider openstack {
         vm concrete_nginx_vm {
            maps nginx_vm
         }
         vm concrete_mysql_vm {
            maps mysql_vm
         }
         net concrete_common_network {
            maps common_network
         }
      }
   }
   active concrete_example1
 }
```

In this layer, the concrete instances of the corresponding abstract infrastructural elements described in the Infrastructure Layer can be found. The chosen provider is OpenStack, as required for this application. Lastly, the concrete infrastructure is set as active.

3.2.3.4 Optimization Layer

The Optimization Layer is shown below:

```
optimization opt {
   objectives {
      "availability" => max
      "performance" => max
   }
   nonfunctional_requirements {
      Req1 "Region" values "00EU" => "region"
      Req2 "Provider" values "openstack" => "provider"
      Req3 "elements" => "VM, VM"
   }
}
```

In the Optimization Layer, it is possible to specify some optimization objectives and non-functional requirements. As required for our application, the maximization of both performance and availability are set as optimization objectives.

Furthermore, as stated in the description of the example, both VMs should be located in the "00EU" region. In order to specify this constraint, a non-functional requirement is defined. Finally, the required provider and the elements we want to deploy as non-functional requirements for the PIACERE Optimizer are defined.

4 Lessons Learnt and Plan for Future Development

The DOML language represents an attempt to develop a high-level modelling approach for Infrastructure as Code that maps on multiple concrete IaC languages.

The experiments carried out so far appear to confirm that the multi-layered approach adopted in DOML is useful as it enables separation of concerns between applications and the underlying infrastructure, but, at the same time, enables an explicit and clear definition of the way applications are mapped on their infrastructures. The possibility to keep the abstract infrastructure definition separated from the concrete one allows users to reuse DOML fragments and map them into different concrete infrastructures.

Keeping the DOML internal representation separated from the external one enables the possibility to conceive different external languages both textual and graphical. While the consolidated version of the DOML is a textual notation, experiments on a graphical Domain Specific Language are being conducted. The objective is to derive the graphical representation from the internal DOML representation by exploiting the libraries and tools made available by the Eclipse platform. Some of these experiments have been reported in Deliverable D3.9 [15, 1, 1] and some others are ongoing work that will be reported before the end of the PIACERE project.

Another important lesson learnt as a result of the latest experiments is that confining crosscutting aspects in a specific layer of the DOML specification is counterproductive. More specifically, at the beginning of the project, for the purpose of parallelizing the research and development work, a simple optimization language has been developed and included in the DOML optimization layer. This particular layer was the only part of the DOML meant to be used by the IOP component to return concrete infrastructural resources fulfilling the optimization objectives and constraints.

In the development of the latest more sophisticated DOML model examples, it has been realized that this approach becomes cumbersome when there are multiple abstract resources, and the user wants to express specific optimization objectives or constraints for each individual resource. In this case, it would have been better to incorporate the optimization aspects within the definition of the resources in the abstract infrastructure layer and to have the IOP to interpret this whole layer to take its decisions. Due to time constraints, at the moment this problem is being addressed in a simplified way by slightly extending the optimization layer language. Improving this part is certainly an important future work.

The extensibility of the DOML is depending on the DOML-E mechanisms defined (see Section 2.1.4). The effectiveness of these messages has to be studied in close connection with the possibility to define in the ICG new templates and modify the existing ones as otherwise any addition into the DOML would not have an impact on the generated IaC code. Given the fact that through the PIACERE project so far, the main focus was the development of the DOML and of ICG, it has not been possible to run extensive experiments in this context. This will be the subject of future work from now till the end of the project and beyond.

As it has been discussed in Section 2.3, all requirements defined for the DOML have been addressed except two that have been partially addressed:

- REQ36 – "*DOML to enable writing infrastructure tests*" is aiming at using the DOML to instrument with proper test cases the canary environment or any other testing environment. An analysis of existing tools in the context has suggested us the possibility to adopt the DOML as input language for the setup of chaos engineering tools, that is,

those tools that introduce failures in a distributed system to test its resilience. Further investigation on this aspect is required and will be the subject for future work.
- REQ57 – "*It is desirable to enable both forward and backward translations from DOML to IaC and vice versa*", was considered, since the beginning of the project, as a very challenging, nice to have, and low priority requirement for the part concerning the backward translation due to two main reasons: the need to have a fully stable DOML language before addressing it and the intrinsic difficulty of deriving a DOML model from a lower level specification. Another critical point is that a DOML specification typically corresponds to IaC code fragments written in different languages. This implies that a single fragment would result in an incomplete DOML model. For the above reasons, it has been decided to limit the backward transformation on moving from the internal DOML representation, serialized in the so called DOMLX, into the DOML external one. Such internal transformation is at the moment under development and will be released at the end of the project with the objective to used it to support transformations between different versions of the DOML.

In addition to the future developments that have been identified above, an important future development will concern the analysis of the role of DOML models with respect to the usage of AI to automatically derive IaC. Recently, projects aiming at generating infrastructural code from natural language are under development [16]. Clearly, such approaches have a significant potential in terms of reducing the time and effort needed to develop IaC. The problem though is the fact that such systems, at least at the moment, are able to deal with specific tasks, such as, "create a Terraform code for provisioning a VM on AWS", but not with the more general task to create all it is needed to support the operation of a complex application.
In this context, the availability of DOML models providing a precise description of the application and infrastructural elements, together with their expected mapping, could potentially help in ensuring that the low level code generated by the AI is correct and complete.

5 Conclusions

This deliverable presents the latest version of DOML (DOML 3.0). Compared to the previous version, it consolidates and cleans up the language and adds specific constructs that were considered critical to address the needs of PIACERE case studies. The deliverable includes a discussion on the fulfilment of requirements, examples of DOML models that demonstrate the modelling details under the new version of DOML, a preliminary analysis of the DOML effectiveness in terms of a comparison with well-known infrastructure languages, and some reflections on the DOML status and on possible future work.

6 References

[1] B. Xiang, E. Di Nitto and G. Novakova Nedeltcheva, “Deliverable 3.2: PIACERE Abstractions, DOML and DOML-E - v2,” PIACERE Consortium, 2022.

[2] PIACERE team, "PIACERE DOML Specification v 3.0.," https://www.piacere-doml.deib.polimi.it/, October 2022.

[3] S. Canzoneri, “DOML v3.0 tutorial,” 2023. https://git.code.tecnalia.com/piacere/public/the-platform/doml/-/tree/main/tutorial

[4] E. Di Nitto ed., “Deliverable D3.1: PIACERE Abstractions, DOML and DOML-E - v1,” PIACERE Consortium, Dec. 2021.

[5] Eclipse Foundation, “Eclipse Modeling Framework,” [Online]. Available: https://www.eclipse.org/modeling/emf/.

[6] Eclipse Foundation, Inc., “Eclipse Xtext™,” [Online]. Available: https://www.eclipse.org/Xtext/.

[7] M. Chiari, B. Xiang, G. Novakova Nedeltcheva, E. Di Nitto, L. Blasi, D. Benedetto and L. Niculut, “DOML: A New Modelling Approach To Infrastructure-as-Code,” in *CAISE 2023*, Zaragoza, 2023.

[8] E. Morganti, “Deliverable 2.2 v 1.1 PIACERE DevSecOps Framework Requirements specification, architecture and integration strategy - v2,” PIACERE Consortium, 2023.

[9] D. Benedetto and L. Nicolut, “Deliverable D3.6: Infrastructural code generation - v3,” PIACERE Consortium, 2023.

[10] Morganti, Emanuele, “Deliverable 2.1: PIACERE DevSecOps Framework Requirements specification, architecture and integration strategy - v1,” PIACERE Consortium, Dec. 2021.

[11] A. Franchini and M. Pradella, “Deliverable D4.3: Verify the Trustworthiness of Infrastructure as Code Task 4.1 Infrastructural Model Verification – v3,” PIACERE Consortium, 2023.

[12] D. E. Iero, “Chaos Engineering: an analysis of processes and tools,” Master Thesis - Politecnico di Milano, 2021.

[13] J. Díaz de Arcaya, “Deliverable 5.3: IaC execution platform prototype - v3,” PIACERE Consortium, 2023.

[14] L. Reboucas de Carvalho y A. Favacho de Araujo, «Performance comparison of Terraform and Cloudify as multicloud orchestrators,» de *CCGRID’20*, 2020.

[15] E. Villanueva, “Deliverable 3.7: PIACERE IDE - v3,” PIACERE Consortium, 2023.

[16] Firefly, “Artificial Intelligence Infrastructure-as-Code Generator,” Firefly, 2023. [Online]. Available: https://aiac.dev/.

APPENDIX: Further details about the DOML

1 Simplified version of the Ericsson case

Being one of the PIACERE use case validation scenarios, the Ericsson case concerns IoT networks in the context of Public Safety applications. The DOML model presented in this section was built as a simplified version of the required infrastructure for the Ericsson case, to be used for validation purposes. The example, per se, does not introduce new DOML concepts compared to the ones exemplified through the case studies in Section 3, but it is more complex than the others as it includes three different networks in the infrastructure layer.

1.1 Case short description

A Public Safety use case network is composed and organized so that dedicated networks can segregate the traffic that flows between the virtual machines to provide an environment that cannot be easily compromised. Therefore, the public safety network includes three different networks for the application components to communicate, and a separate network for operation and management.

In this example, three different software components are deployed on three different virtual machines, provisioned by the OpenStack cloud provider. Besides such software components, the application needs to access two different Software-as-a-Service components providing APIs, accessible through specific URL.

Finally, the infrastructure should be configured in such a way to guarantee maximum availability and performance while minimizing the cost. There are two non-functional requirements: the maximum cost should be 300.0, while the minimum performance measure should be 7.0%.

1.2 DOML model

```
doml uc3_openstack

application app {

   software_component iwg {

      provides { net_info }

   }

   software_component osint {

      provides { osint_info }

      consumes { net_info, get_twitter, ewcf_rest_interface }

   }

   software_component ewcf {

      provides { ewcf_rest_interface }

      consumes { get_firebase }

   }

   saas external_twitter {

      provides { get_twitter @ "https://twitter_api/get" }

   }
```

```
  saas external_firebase {

    provides { get_firebase @ "https://firebase_api/get" }

  }

}

infrastructure infra {

  // VMs region

      //  This  VM  as  a  network  interface  belonging  to  the  OAM  network,
      //   one   belonging   to   Net1   to   communicate   with   the   osint   VM
      // and one belonging to Net2 to access the 5G network

  vm igw_vm {

    os "Ubuntu-Focal-20.04-Daily-2022-04-19"

    size "small"

    iface igw_vm_oam {

      belongs_to subnet_oam_igw

    }

    iface igw_vm_net1 {

      belongs_to subnet_net1_igw

    }

    iface igw_vm_net2 {

      belongs_to subnet_net2_igw

    }

    credentials ssh_key

  }

      //  This  VM  has  a  network  interface  belonging  to  the  OAM  network,
      //   one   belonging   to   Net1   to   communicate   with   the   other   VMs
      // and one belonging to Net3 to the access the Internet

  vm osint_vm {

    os "Ubuntu-Focal-20.04-Daily-2022-04-19"

    size "small"

    iface osint_vm_oam {

      belongs_to subnet_oam_osint

    }

    iface osint_vm_net1 {

      belongs_to subnet_net1_osint

    }

    iface osint_vm_net3 {

      belongs_to subnet_net3_osint
    }

    credentials ssh_key

  }

     //  This  VM  has  a  network  interface  belonging  to  the  OAM  network,
     //   one   belonging   to   Net1   to   communicate   with   the   osint   VM
     // and one belonging to Net3 to access the Internet

  vm ewcf_vm {

    os "Ubuntu-Focal-20.04-Daily-2022-04-19"

    size "small"

    iface ewcf_vm_oam {

      belongs_to subnet_oam_ewcf

    }

    iface ewcf_vm_net1 {

      belongs_to subnet_net1_ewcf

    }

    iface ewcf_vm_net3 {

      belongs_to subnet_net3_ewcf

    }

    credentials ssh_key

  }

  // Operation and management network, to which all the VMs are connected

  net oam {

    protocol "TCP/IP"

    cidr "16.0.0.0/24"

    subnet subnet_oam_igw {

      protocol "TCP/IP"

      cidr "16.0.1.0/26"

    }

    subnet subnet_oam_osint {

      protocol "TCP/IP"

      cidr "16.0.1.64/26"

    }

    subnet subnet_oam_ewcf {

      protocol "TCP/IP"

      cidr "16.0.1.128/26"

    }
```

```
}


// This is an internal network and therefore has no Internet gateway
net net1 {
  protocol "TCP/IP"
  cidr "16.0.1.0/24"
  // Subnets definition
  subnet subnet_net1_igw {
    connections {
      subnet_net1_osint
    }
    protocol "TCP/IP"
    cidr "16.0.1.0/25"
  }
  subnet subnet_net1_osint {
    connections {
      subnet_net1_igw,
      subnet_net1_ewcf
    }
    protocol "TCP/IP"
    cidr "16.0.1.64/26"
  }
  subnet subnet_net1_ewcf {
    connections {
      subnet_net1_osint
    }
    protocol "TCP/IP"
    cidr "16.0.1.128/26"
  }
}
// Network connecting igw to 5G
net net2 {
  protocol "TCP/IP"
  cidr "16.0.2.0/24"
  subnet subnet_net2_igw {
    protocol "TCP/IP"
    cidr "16.0.2.0/25"
  }
  gateway net2_igw {
    address "16.0.2.22"
  }
}
// Network connecting osint and ewcf to Internet
net net3 {
  protocol "TCP/IP"
  cidr "16.0.3.0/24"
  subnet subnet_net3_osint {
    protocol "TCP/IP"
    cidr "16.0.3.0/25"
  }
  subnet subnet_net3_ewcf {
    protocol "TCP/IP"
    cidr "16.0.3.128/25"
  }
  gateway net3_igw {
    address "16.0.3.22"
  }
}
//Credentials region
// These credentials are used to access the VMs
key_pair ssh_key {
  user "ubuntu"
  // key to be inserted here
  keyfile "…"
  algorithm "RSA"
  bits 4096
}
// Security region
    // This security group is composed of standard security rules
    // and is associated with all the interfaces
```

```
   security_group sg {
      egress icmp {
         protocol "ICMP"
         from_port -1
         to_port -1
         cidr ["0.0.0.0/0"]
      }
      ingress http {
         protocol "TCP"
         from_port 80
         to_port 80
         cidr ["0.0.0.0/0"]
      }
      ingress https {
         protocol "TCP"
         from_port 443
         to_port 443
         cidr ["0.0.0.0/0"]
      }
      ingress ssh {
         protocol "TCP"
         from_port 22
         to_port 22
         cidr ["0.0.0.0/0"]
      }
      ifaces igw_vm_oam, igw_vm_net1, igw_vm_net2, osint_vm_oam, osint_vm_net1,
osint_vm_net3, ewcf_vm_oam, ewcf_vm_net1, ewcf_vm_net3
   }
}
deployment config1 {
   osint => osint_vm,
   iwg => igw_vm,
   ewcf => ewcf_vm
}
active deployment config1
```

```
concretizations {
   concrete_infrastructure con_infra {
      provider openstack {
         // Concrete computing nodes region
         vm concrete_osint_vm {
            maps osint_vm
         }
         vm concrete_igw_vm {
            maps igw_vm
         }
         vm concrete_ewcf_vm {
            maps ewcf_vm
         }
         // Concrete networks region
         net concrete_oam {
            maps oam
         }
         net concrete_net1 {
            maps net1
         }
         net concrete_net2 {
            maps net2
         }
         net concrete_net3 {
            maps net3
         }
      }
   }
   active con_infra
}
optimization opt {
   objectives {
      "cost" => min
      "performance" => max
      "availability" => max
   }

   nonfunctional_requirements {

      req1 "Cost <= 300" max 300.0 => "cost"

      req2 "Performance >= 7%" min 7.0 => "performance"

      req3 "Provider" values "openstack" => "provider"

      req4 "elements" => "VM, VM, VM"

   }

}
```

2 User manual to extend DOML

This brief manual has the purpose to help users in the process of applying modifications to the DOML language to extend it.

2.1 Required tools

First, it must be pointed out that the PIACERE framework and DOML rely on the open-source Eclipse IDE.

The Eclipse IDE version used to build DOML v3.0 is “Eclipse 2023-03”: in particular, it is used the ***Eclipse Modeling Tools*** package, which can be selected directly from the installer.

Such a package already includes ***Eclipse Modeling Framework (EMF)*** installed with it. As reported in the official website [5], the EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. This framework has been used to build the DOML Ecore, the formalization of its metamodel.

To make the metamodel more readable and easily accessible to programmers, we have made use of ***Eclipse Emfatic***, a textual syntax for EMF Ecore metamodels. The syntax is very intuitive and easy-to-use.

It is strongly recommend installing the Emfatic plugin to make changes to the DOML metamodel.

In order to generate the Ecore from the Emfatic code and to being able to generate graphical models from it, the ***Eugenia*** tool, which is part of the ***Eclipse Epsilon*** plugin, is required. The metamodel diagrams shown in the D3.3 document have been generated using Eugenia.

The DOML syntax has been generated by using ***Eclipse Xtext***, which is a framework for development of programming languages and domain-specific languages, as reported in the official website [6].

Users will, therefore, need to install the Xtext plugin to apply changes to the syntax.

2.2 Metamodel update

Steps to update the metamodel:

1. Open the implementation folder as workspace inside the Eclipse IDE.
2. Open the “eu.piacere.doml” project.
3. Inside the “model” folder, open the “doml.emf” file and apply the needed changes.
4. After saving the file, right click on it in the Project Explorer and in the “Eugenia” sub-menu, select “Generate EMF Editor”.

2.3 Syntax update

Steps to update the syntax:

1. Open the implementation folder as workspace inside the Eclipse IDE.
2. Open the “eu.piacere.doml.grammar” project.
3. Inside the “src/eu.piacere.doml” folder, open the “Doml.xtext” file and apply the needed changes.
4. After saving the file, right click on it in the Project Explorer and in the “Run as” sub-menu, select “Generate Xtext Artifacts”.

2.4 Testing applied changes and updating the ICG

To test the DOML editor after the modifications have been applied, right click on the "eu.piacere.doml" project in the Project Explorer (Plug-in Development perspective) and in the "Run as" sub-menu, select "Eclipse Application". At this point, after the application is launched, try creating a new project containing a new file with the ".doml" extension: here the user can test the editor, verifying that the changes to the syntax have been applied correctly.

When extending DOML, it is fundamental to remember that not only the metamodel and the syntax must be updated, but also the Infrastructure Code Generator (ICG), which interprets the DOML language and generates the corresponding code in IaC standard languages. To see how to deal with the changes to be applied to the ICG, please refer to the User Manual in the Deliverable [9].