PIACERE

Deliverable D3.2

PIACERE Abstractions, DOML and DOML-E – v2

Editor(s):	Bin Xiang, Elisabetta Di Nitto, Galia Novakova Nedeltcheva
Responsible Partner:	Politecnico di Milano/ PoliMi
Status-Version:	Final - v1.0
Date:	30.11.2022
Distribution level (CO, PU):	Public

Project Number:	101000162
Project Title:	PIACERE

Title of Deliverable:	PIACERE Abstractions, DOML and DOML-E – v2
Due Date of Delivery to the EC	30.11.2022

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):	Eliseo Villanueva Morte (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 latest consolidated version of the DOML the metamodel and the corresponding semantic, as well as machine-readable descriptions of the aspects that are relevant to the main phases of the IaC lifecycle seamlessly integrated with the design and development of the IaC lifecycle. This metamodel will be then presented as an end-user language enabling the modelling of the different elements needed for the infrastructure provisioning, configuration management and deployment, the deployable infrastructural components, constraints and so on. The various iterations will seek and take into consideration the feedback from PIACERE's end users and, possibly, other users outside the project to ensure that the language is sufficiently powerful and simple to use.

Keyword List:	Model-driven engineering, metamodels, modelling abstractions, Infrastructure as Code

Document Description

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
v0.1	29.09.2022	First draft version	PoliMi
v0.2	20.10.2022	Comments and suggestions received by consortium partners	ALL partners
v0.3	02.11.2022	Second draft version ready for review	Bin Xiang (PoliMi)
v.0.4	22.11.2022	Reviewed version	Eliseo Villanueva Morte (Prodevelop)
v0.5	23.11.2022	Final draft after the review	PoliMi
v1.0	25.11.2022	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 deliverable provides updates to deliverable D3.1 [1], including the progress of the DOML development to fulfil the requirements during the evolution of the project.

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 PIACERE KR1). DOML models are created using the PIACERE IDE (PIACERE KR2), which provides the users the guidance and 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.

In this deliverable, we first introduce the DOML release plan that has been defined in order to have clear control of the development and release. Specifically, a committee has been set up to define the schedule of each release and make decisions about the features to be added in each release. The committee receives requests for update from its members or from other DOML users.

Furthermore, we present the status of requirement fulfilment, the main changes concerning the DOML metamodel and the DOML extensions, and the corresponding changes of DOML examples. The changes are made when introducing new functionalities and fixing the problems raised during the development. The deliverable is also accompanied by a new version of Annex [2] that provides a detailed definition of all concepts of the DOML and that is released as a separate document to facilitate its usage and evolution independently of this deliverable D3.2.

In the line of our continuous analysis of the state of the art, we compare also the DOML with the related approach EDMM [3], and highlight the novelties of one with respect to the other.

Finally, based on the status of requirement fulfilment and possible functional improvements, we summarize the future plan for development of DOML and draw the conclusions of this deliverable.

1 Introduction

This deliverable describes the current status of DOML at M24. It provides updates to the previous deliverable D3.1 [1], describing the main developments of DOML throughout the second year of the project.

In the second year, the main activities are focused on the revision of DOML metamodel to clarify some of the confusing concepts, and functionality improvement of DOML for enhancing the capability of expressing various software and infrastructure elements. These new updates are based on the DOML requirements defined at the beginning and the new ones raised during the testing on examples and case studies and the integration with other PIACERE tools, e.g., IaC Code Generation (ICG), Model Checker (MC), etc.

Meanwhile, to facilitate the development DOML, we also introduce the DOML release plan, which presents the ways for the users' community to provide feedbacks and issues when using the modelling language and presents the procedure of dealing with all these suggestions and issues, and release of new version of DOML by the DOML committee during the evolution of the project. Currently, the history of DOML version includes the prototype v0.1, the advanced v2.0 and the improved v2.1.

Finally, DOML is compared with a similar approach named EDMM from different aspects, including the target activity, metamodel, extensibility, code generation, etc. to highlight the novelty and advantage of DOML.

1.1 About this Deliverable

The main goals of this deliverable are to: 1) introduce the DOML release plan for better management of DOML development; 2) present the main changes introduced in DOML 2.1 [4] based on the DOML requirements; 3) compare the differences between DOML and the Essential Deployment Metamodel (EDMM) [3] approach.

1.2 Document Structure

The document is organized as follows:

- Section 2 illustrates the DOML release plan.
- Section 3 describes the fulfilment status of DOML requirements in terms of general characteristics and specific elements.
- Section 4 presents the scenarios that have been defined for DOML in the second project year.
- Section 5 presents DOML 2.1 metamodel, focusing on the main changes.
- Section 6 points to the DOML extension mechanism.
- Section 7 describes two examples modelled with DOML 2.1.
- Section 8 compares the differences between DOML and the Essential Deployment Metamodel (EDMM) approach.
- Section 9 summarizes the plan for future development.
- Section 10 concludes deliverable D3.2.

The deliverable is accompanied by Annex 1 [2] which includes the detailed specification of the DOML concepts.

2 Overview of the DOML Release Plan

This section describes the means for the DOML users' community to provide suggestions and report issues in the current status of the proposed modelling language. Moreover, it presents the procedure by which all these suggestions and issues will be handled both during the PIACERE project and after the research has been completed. An important element of the release plan is the DOML committee that is in charge of collecting and analysing the incoming suggestions and issues, of prioritizing them and of planning the work needed to address them.

The DOML Committee will be the main body in charge of updating and releasing new versions of the DOML language during the PIACERE project and after the project has finished. The committee is composed of a chairperson and a set of two or more experts, ideally from different organizations. The goal of the committee is to receive, sort and resolve all incoming issues and suggestions for changes to the current DOML version, and provide a proper solution to them, either by responding to the suggestion/issue or by including a modification to the DOML version to be released at the next release date.

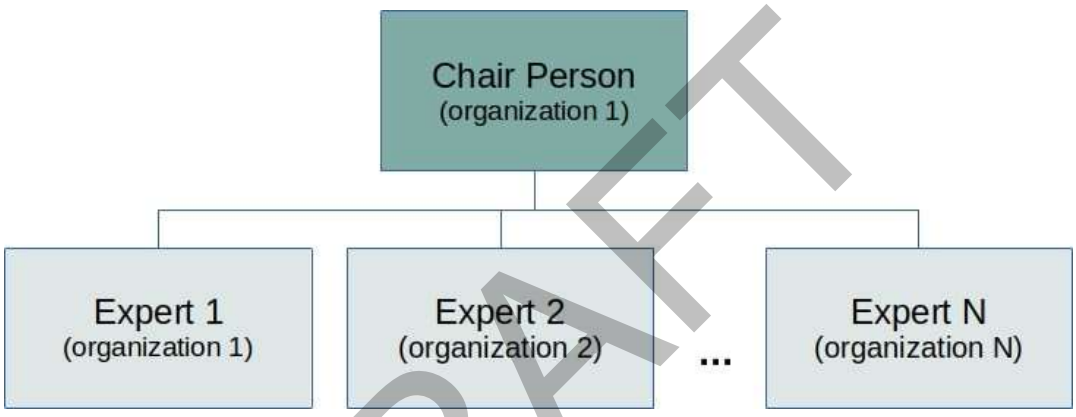

Figure 1. DOML Committee composition

Periodicity of committee meetings

The meetings of the DOML committee are biweekly during the PIACERE project, and monthly after PIACERE ends. The goal of these periodic meetings is to ensure that the greatest number of issues are solved before the next version of DOML is released.

Roles of the members of the committee

There are two different roles in the DOML committee:

Chair Person. The Chair Person is responsible of scheduling the committee meetings and ensuring that all the issues and suggestions received are sent to the group of experts for review. They are also in charge of planning the issues that will be discussed during every meeting. Finally, the Chair Person will act as the owner of the issues backlog, prioritizing pending tasks, and assigning them to the experts.

Expert. A DOML expert is responsible of reading and evaluating all the issues selected for review in the upcoming DOML committee meetings. They are also responsible of providing solutions to the issues/suggestions assigned to them, as well as integrate any suggestions from their colleague experts to the draft solution.

Lifecycle of Issues/Suggestions

The activity diagram in Figure 2 describes the envisaged lifecycle for all incoming issues and suggestions.

Firstly, received issues and suggestions are discussed which can be either approved or added to the backlog, or rejected. In case of rejection, the authors of the rejected issue or suggestion should receive a response from the committee, describing the rationale behind the rejection.

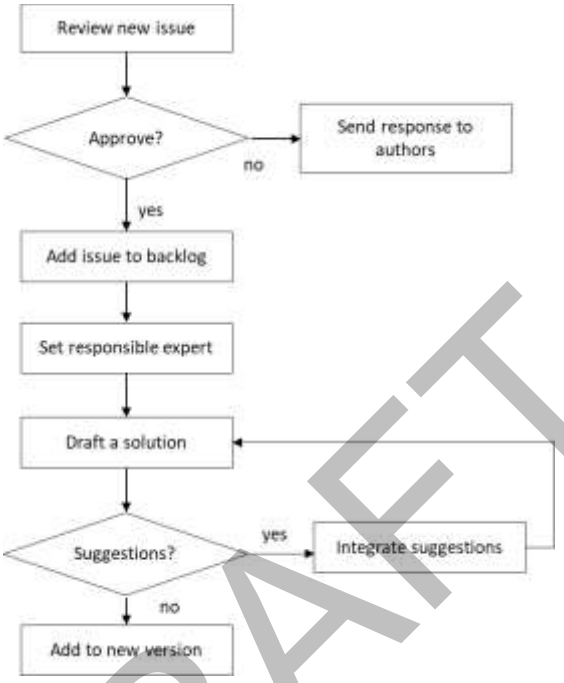

Figure 2. Lifecycle of issues

Once the issues are in the backlog, the Chair Person should prioritize them (assisted by the experts) and then assign the ones with the highest priority to the experts.

An issue that has been assigned to an expert must then wait for a draft solution to be issued. The draft solution must be then shared with the rest of the expert's panel and the solution is then set to be reviewed. The expert in charge of the issue will be responsible of integrating the contributions by the other members of the committee and provide the final version for approval to the Committee.

After the final approval, the change is set to be included in the next version of the DOML. In case the Committee does not approve the solution, then the Chair Person will select a different expert, and a new solution will be discussed as explained before.

Description of an Issue

In order for the committee to correctly understand and address issues and suggestions submitted for review, the following template is proposed (see Table 1). The template will be made available to the DOML community to facilitate the submission of inputs for the DOML.

DOML Release Schedule

Each version of DOML will include all the issues and suggestions whose solutions have been approved by the DOML Committee. The schedule of new releases is agreed by the DOML

Committee with the other PIACERE project members. The main criterion is that a new release is issued every time a critical set of issues has been addressed.

Releases in months M24 and M36 shall be considered major, while the rest of the releases during PIACERE shall be considered as minor.

The steps to be performed in order to release a new version of the DOML are the following:

- The content of the DOML main repository is cleaned up and aligned. In particular, the DOML specification is modified to reflect the changes in the DOML metamodel and in the syntax.
- The new DOML specification document is published on the DOML website (https://www.piacere-doml.deib.polimi.it/).
- The test examples are updated to reflect the changes in the DOML metamodel and syntax.

The whole content of the DOML repository is tagged with the new version number.

Table 1. Definition of an issue/ suggestion in DOML

Field	Description
ID	Automatically assigned number. Used to univocally identify and issue/suggestion.
Kind	Issue or Suggestion. Issues should address problems in the current specification, while suggestions should address recommendations for improving the current version of DOML.
Title	A descriptive title for the issue/suggestion.
Description	A thorough description of the issue or suggestion, including, where necessary, examples.
Proposed solution (optional)	A proposed solution to the issue/suggestion. The committee shall use this solution as starting point; however, there is no obligation for the committee to accept it.
Criticality (optional)	Blocking/High/Medium/Low. The level of criticality from the point of view of the DOML user. The committee may change the level of criticality of the issue/suggestion after approval, thus, this is just a hint for the committee.

3 Status of DOML Requirements Fulfilment

The requirements for the DOML are presented as part of Deliverable D2.1 [4], and summarized in Deliverable D3.1 [1]. In this section we provide an updated summary of the DOML-specific requirements status at M24 grouped in two tables, one listing the requirements 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 how the requirement is addressed or planned to be addressed is provided. 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 within the analysis carried out as part of Deliverable 2.2 [5] have been kept.

In the follow tables, the rows highlighted in green indicate the corresponding DOML requirements that have been fulfilled so far. The remaining rows correspond to requirements that are partially addressed.

Table 2. Requirements on the general characteristics of DOML [2].

Req ID	Description	How DOML is addressing this requirement at M24
REQ63	DOML must be unambiguous.	DOML is formally defined in terms of its translation into the corresponding IaC code fragments. As such, it is not ambiguous by definition. It has been addressed in year one. For the new DOML elements that have been introduced within year two a translation into IaC has been defined as well.
REQ62	DOML must support different views.	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. It has been addressed in year one.
REQ70	The DOML should allow users to state correctness properties in a suitable sub-language	This requirement is addressed in the DOML in two different ways: 1) The main correctness relationships among elements in the specification are offered directly as part of the language and can be used. This has been done in year one.

	(possibly Formal Logic).	2) The language offers the possibility to express generic constraints on the elements that are used in a certain DOML model. This has been done in year two.
REQ76	DOML should allow the user to model information needed for each of the four considered DevOps activities (Provisioning, Configuration, Deployment, Orchestration)	The requirement has been rephrased in year two to clarify that the DOML does not describe explicitly the operations provisioning, orchestration etc., but it provides pieces of information that are relevant to these phases. The rephrased requirement has been partially addressed in year one. The level of support has been improved in year two and is planned to be further improved in year three.
REQ57	It is desirable to enable both forward and backward translations from DOML to IaC and vice versa	DOML currently supports to the forward translation to different IaC, e.g., Terraform and Ansible. This has been done in year one. The translation to other IaCs is still an ongoing work. 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 will try to address backward translations in the last part of the project when the DOML will be consolidated.

Table 3. Requirements on the specific elements to be modelled in DOML

Req ID	Description	How DOML is addressing this requirement
REQ01	The DOML must be able to model infrastructural elements.	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. This implies that the DOML will offer the possibility to define new elements through the extension mechanisms (DOML-E). The requirement is fulfilled in year one, and in year two, new infrastructural concepts are introduced.
REQ25	DOML should support the modelling of security rules (e.g., by type tcp/udp..., and ingress/egress port definition)	This requirement is fulfilled by the new concept of security group introduced in year two, which contains both ingress and egress security rules.
REQ26	DOML should support the modelling of security	This requirement is addressed by a specific construct in the language. This has been done in year one.

	groups (containers for security rules)	
REQ27	DOML should support the modelling, provisioning, configuration, and usage of container engine execution technologies (e.g., docker-host)	The DOML addresses this requirement by offering constructs to define a container, a container image, and a container file. This has been done in year one.
REQ28	DOML should support the modelling of containerized application deployment (e.g., pull/run/restart/stop docker containers)	This requirement is partially addressed. As stated for REQ27, the DOML offers the possibility to model containers and its constituents in year one, and the concept is improved for the configuration of container in year two. As stated for REQ76, the DOML does not support the explicit modelling of workflows to which the pull/run/restart/stop activities belong to. It, however, supports the possibility to link to DOML elements script files together with their interpreters. This opens up the possibility to define such 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	This requirement is fulfilled in year two with the possibility to support different platform for VM provisioning.
REQ30	DOML should enable support for policy definition constraints for QoS/NFR requirements	This requirement is partially addressed. DOML supports the definition on QoS/NFR requirements (see REQ61). We will assess whether the definition of such requirements is sufficient to actuate the policies defined by the PIACERE runtime components or whether additional and specific policies will have to be defined as part of the language.
REQ58	DOML should offer the modelling abstractions to define the outcomes of the IoP	This requirement is fulfilled in year two by introducing the optimization solution concept which is composed of results of the objectives and the decision variables.
REQ59	The DOML should allow users to define rules and constraints for redeployment, reconfiguration, and other mitigation actions	This requirement is partially addressed. 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. This has been done in year one and improved in year two. Redeployment rules are to be addressed.
REQ60	DOML should support the modelling of security metrics both at the level of infrastructure and application	For the moment, this aspect is not explicitly addressed in the DOML. The sub-language used for defining generic non-functional requirements could be suitable to address the modelling of security metrics as well. Experiments will be conducted, and the language will be extended if needed.
REQ61	DOML must support the modelling of NFRs and of SLOs	NFRs and SLOs have been supported in year one, and in year two, functional and non-functional requirements have been separated and supported, where functional requirements fulfilment is checked by the verification tool, and the non-functional requirements are mainly used to describe the constraints for the IOP (infrastructure optimization).
REQ36	DOML to enable writing infrastructure tests.	This requirement is not addressed in the DOML. 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 is ongoing and will provide inputs to address this requirement.

4 Newly defined scenarios

This section presents the new scenarios of DOML usage that have been specifically analysed and addressed in the second project year. The structure of these scenarios is aligned with the guidelines associated to agile development using Gherkin syntax. They concern, in particular, the following aspects:

- Associate a software component to a container.
- Define the container as part of the infrastructure.
- Associating a software component to specific IaC code.
- Creating an autoscaling group.
- Defining functional and non-functional requirements.
- Extending the DOML with new resources/providers.

```
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

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

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

Scenario: Associate a software component to a container
Given the software_component portainer
When user digits something like:
repo r1 {
  engine 'docker'
  uri "hub.docker.com/r/portainer/portainer"
  user_pass u1 {
    user "***"
    pass "***"
  }
}
Then the software_component portainer is associated to the image defined by
the uri

Scenario: Define the container as part of the infrastructure
Given a DOML model
When user digits something like:
software_component c1 {
  source_uri "..."
}
...
container co1 {
  repo r1 {
    engine 'docker'
    uri "https://hub.docker.com/_/httpd"
    user_pass u1 {
      user "***"
      pass "***"
```

```
    }
  }
  destination_dir "/usr/local/apache2/htdocs/"
}

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

deployment config {
  c1 -> co1
  co1 -> vm1
}
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

Scenario: Associating a software component to specific IaC code
Given a DOML model
When a user digits something like the following
software_component nio3_git {
  repo r1 {
    engine 'git'
    uri "git.code.tecnalia.com/piacere/private/wp7-use-cases/uc1.si-mpa/-
/tree/main/deploy-nio"
    entry "ansible/provision.yml"
    backend "ansible"
    user_pass u1 {
      user "***"
      pass "***"
    }
  }
  properties {
    nexus_docker_registry_user = "***";
    nexus_docker_registry_password = "***";
  }
}
Then software_component nio3_git relevant code is found in the specified URI.
And The code is meant to be executed starting from the specified entry, with
the specified backend (Ansible in the example)


Scenario: Creating 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 an 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


Scenario: Defining 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
```

**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
```

5 DOML 2.1 Metamodel

The DOML metamodel is constructed by 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.

5.1 Main Changes

Compared to the first version of DOML metamodel in D3.1 [1], DOML 2.1 metamodel mainly contains the following changes:

- **Commons Layer:**
  - DOML version is added for the identification of DOML that will be used in different tools, e.g., ICG, MC, etc.
  - Different types of properties are supported for the convenience of modelling.
  - Requirements are split into functional and non-functional ones.
- **Infrastructure Layer:**
  - Security group is implemented, supporting definition of ingress and egress security rules for the network.
  - Autoscaling group is implemented, supporting definition of VM template and load balancer.
  - VPC is removed, the more general concept Subnet is implemented.
  - CIDR field is added in abstract network to define IP ranges.
  - Container configuration is added to define the port mapping, environment variables, etc.
  - Credentials for VMs are introduced, including SSH key pair and user password.
  - Docker swarm is introduced to support/define the cluster of Docker container services.
  - Label and size fields are added in storage.
  - CPU, memory, size, etc. fields are updated (with explicit units).
- **Concrete Layer:**
  - Data structure is changed, i.e., the runtime provider now has containment references of the other concrete elements.
  - IP address is moved from the abstract network to the concrete one.
  - VM and container images are added.
- **Optimization Layer:**
  - Non-functional requirements are added to define the constraints of the optimization problem.
  - Optimization solutions are added to store the results by the IOP component.

For the sake of completeness, Sections from 5.2 on describe the whole metamodel structured in multiple layers.

5.2 Commons Layer

The **Commons Layer** contains the main abstract application agnostic concepts that are shared among different layers (see Figure 3). 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.

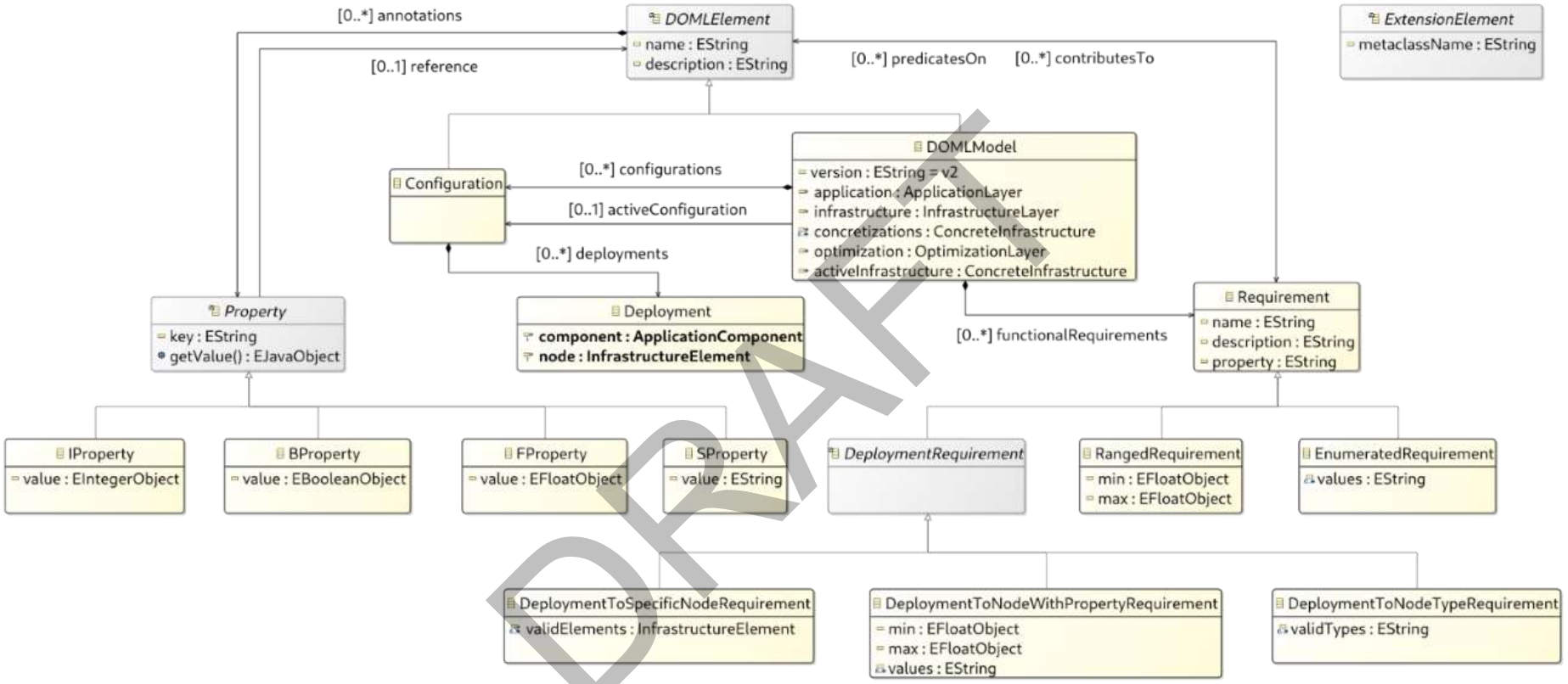

Figure 3. Commons Layer diagram

5.3 Application Layer

The **Application Layer** (see Figure 4) 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.

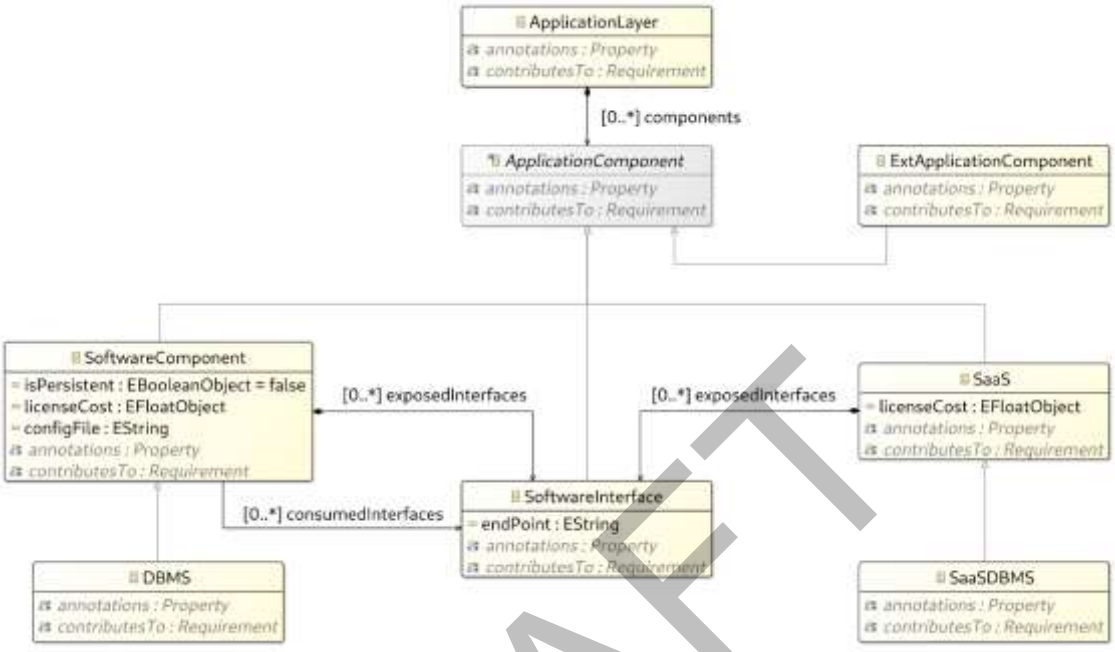

Figure 4. Application Layer diagram

5.4 Infrastructure Layer

The **Infrastructure Layer** (see Figure 5 for an overview. The readability of this figure could not be improved for lack of space. A detailed description of all its elements is available in the Annex [2] defines the abstract infrastructure elements that will be used to deploy the application components. 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, be it a VM from AWS or a specific VM deployed by the user.

5.5 Concrete Layer

The **Concrete Layer** (see Figure 6) 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.

5.6 Optimization Layer

The **Optimization Layer** (see Figure 7) defines all the information required for the optimizers to locate the best configurations for the cloud applications described in the DOML, as well as means to capture the optimization solutions.

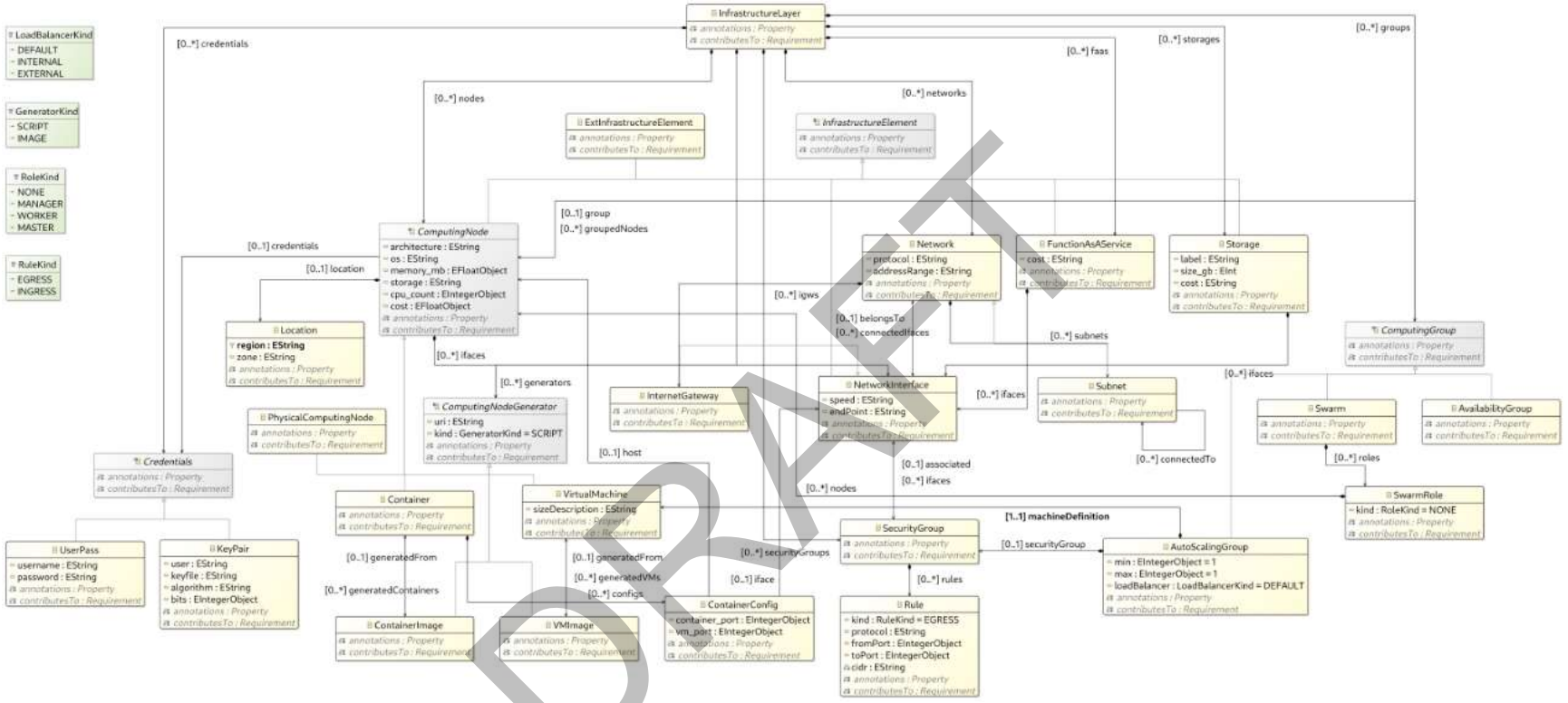

Figure 5. Infrastructure layer diagram

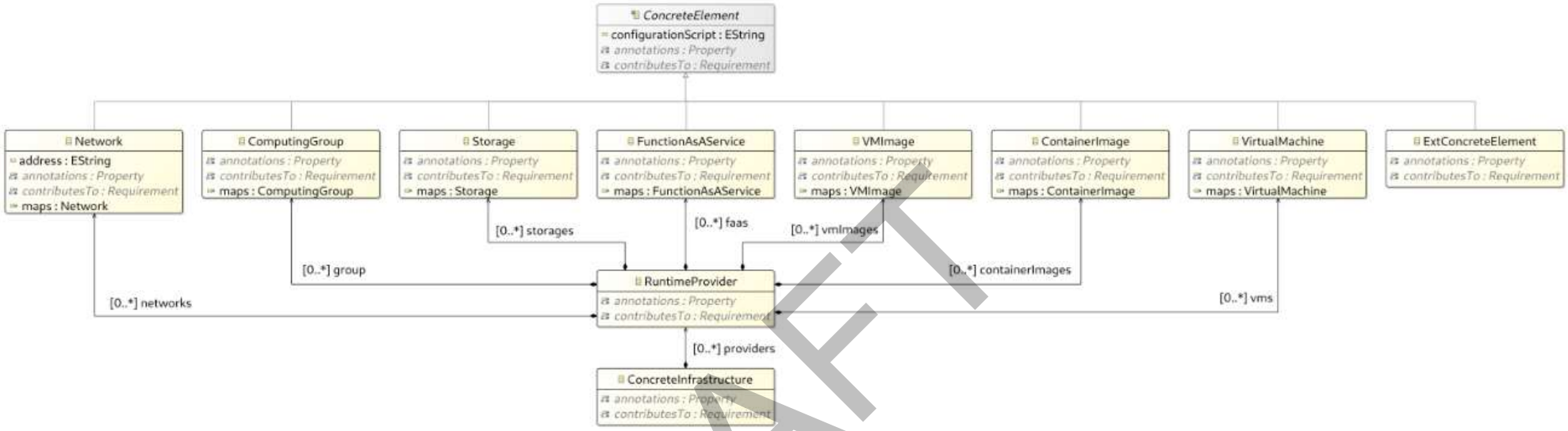

Figure 6. Concrete Infrastructure Layer diagram

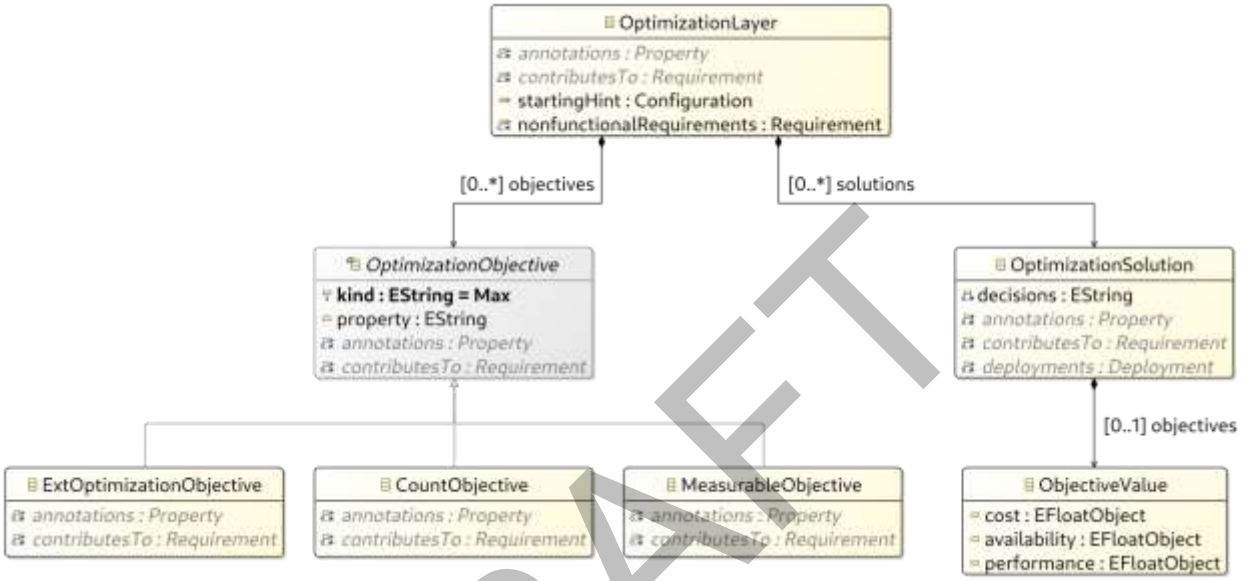

Figure 7. Optimization Layer diagram

6 DOML Extension Mechanism (DOML-E)

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

In the following, we illustrate the detailed steps for extending DOML in the above two ways through examples. For extending metamodel, we have defined a new scenario in year two for this deliverable (see Section 4).

6.1 Creation of New Concepts

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

To this end, we will first modify the metamodel by creating a metaclass named Service which extends the abstract class DOMLElement and includes the necessary attributes and references. For the sake of simplicity and clarity, we create the following example Service:

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

In the above example, a Service is an 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;
      ref ComputingNode [*] hosts;
}
```

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

Now we are ready to create the corresponding concrete syntax for them. For the service, we could use 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)* ']' )?&
      ('properties' '{'
         annotations+=Property*
      '}')?
```

```
   )
;
```

Since we can reuse the Container concept, we update it by adding the characteristics as a service:

```
Container returns infra::Container:
   'container' (('service' DOMLElement '{' Service) | DOMLElement '{')
      'hosts' hosts+=[infra::ComputingNode] (','hosts+=[infra::ComputingNode])*
   '}'
;
```

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 {
   hosts vm_infra
   port 53
   constraints [ 'C1', 'C2' ]
   //image: "nexus-registry.xlab.si:5001/consul:1.0.0"
}
```

6.2 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 extends 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 show an example of integer property definition. Other properties like string, float, etc. are defined in the similar way.

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.

7 DOML 2.1 Example

In this section, we reflect the main changes and functionalities of DOML 2.1 for the examples available in D3.1 [1], i.e., the WordPress website and the Function-as-a-Service (FaaS) Thumbnail Generator.

7.1 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 the 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. In the following, for the sake of clarity, we only demonstrate the new code fragments that implement this application with DOML v2.1, where the rest is the same as the one described in the previous version shown in D3.1 [1]. Specifically, the updates include the definition of the container with detailed the configuration, the security group composed of different ingress and egress rules, the network and subnet, the credentials and the autoscaling group. In DOML v2.1, some of these concepts are revised and the others are newly introduced.

For modelling the **docker container**:

```
container container1 {
      host wp_vm1 {
            container_port 80
            vm_port 80
            iface i1
            properties {
                  WP_DB_HOST = "dbms_vm";
                  WP_DB_USER = "username";
                  WP_DB_PASSWORD = "password";
                  WP_DB_NAME = "database.name";
            }
      }
}
```

In this fragment, "container1" is to be hosted on the virtual machine "wp_vm1" with the port mapping "80:80" binding with network interface "i1". The properties, i.e., "WP_DB_HOST", "WP_DB_USER", "WP_DB_PASSWORD" and "WP_DB_NAME" are provided as the environment variables for the container.

For modelling the **security group**:

```
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"]
      }
}
```

The security group defines several egress rules (w.r.t. ICMP) and ingress rules (w.r.t. HTTP, HTTPS and SSH) for the network.

For modelling the **network** and **subnet**:

```
net net1 {
      cidr "/24"
      protocol "tcp/ip"
      subnet subnet1 {
            cidr "/24"
            protocol "tcp/ip"
      }
}
```

This fragment defines the CIDR and protocol for the abstract network and subnet.

For modelling the **credentials**:

```
user_pass ssh_pass {
      user "username"
      pass "password"
}
key_pair ssh_key {
      keyfile "ssh key"
}
```

The credentials (including both password and ssh key) for the virtual machine can be defined.

For modelling the **autoscaling group**:

```
autoscale_group ag {
      vm wp_vm {
            cpu_count 2
            mem_mb 1024.0
            iface i1 {
                  belongs_to net1
                  security sg
            }
            credentials ssh_pass
      }
      // count = 2
      min 2 max 2
}
```

The autoscaling group is defined by providing the template of virtual machine “wp_vm” and the minimum and maximum number of VMs supported.

7.2 FaaS Thumbnail Generator

The second example implements an online thumbnail generator based on a Function-as-a-Service (FaaS) infrastructure. The generator works in this way: the user uploads the high-resolution image they want to generate the thumbnails for, and then the service resizes it to

three different sizes, which are then made available to the user. The image-resizing functionality is implemented as a stateless FaaS service.

The structure of the FasS Thumbnail generator is that: a web interface is provided for the user to upload and download images, and a FaaS software is used to do the image resizing task for producing thumbnail; two storage buckets (e.g., based on provider AWS) are used to store respectively the input and output images; a notification service (as a software component) manages the communication between the web app and the resizing function.

In the following, for the sake of clarity, we only demonstrate the new code fragments that implement this application with DOML v2.1, where the rest is the same as the one described in the previous version shown in D3.1 [1]. Specifically, the main changes are related to the definition of VM, which states the network and interface, the security group, the credentials, and the location information.

For modelling the **VM**:

```
vm v {
      iface i1 {
            address "10.0.0.1"
            belongs_to net1
            security sg
      }
      credentials ssh_key
      loc {
            region "eu-central-1"
      }
}
```

The security group, credentials and location information are added in for the VM. The detailed changes on network, subnet, security group, credentials, etc. are similar to the ones for WordPress example in Section 7.1.

8 Comparison between DOML and Essential Deployment Metamodel (EDMM)

EDMM is a project started by University of Stuttgart [3]. It stands for Essential Deployment MetaModel and defines the main concepts that enable a comparison between different deployment IaC frameworks. Besides the metamodel, transformators are being built to generate code in various IaC languages.

In Table 4 below we compare the EDMM with our DOML + ICG approach.

Table 4. Comparison between EDMM+Transformator and DOML+ICG

Topic	EDMM+Transformator	DOML+ICG
Target activity	Deployment	Provisioning, Configuration, Deployment, Orchestration
Metamodel	Very abstract base metamodel, domain-specific types are defined in the model (e.g. based on the provided type.yaml) and are extensible	Domain-specific metamodel, domain-specific types are already defined and fixed
Extensibility	The metamodel is fixed, but contains both type and property meta-elements to define new types in the model. All domain-specific types are defined as extensions in each model (e.g. see types.yml)	DOML-E extension mechanism should allow to define new meta-classes to extend the meta-model. The metamodel has the property meta-element, plus specific meta-elements, subclasses of ExtensionElement, whose instances will extend the meta-model
Code generation strategy	Template Based Code Generation (TBCG)	TBCG
Code Generator architecture	Model Parser + one Plugin for each target IaC language	Model Parser + one Plugin for each target IaC language
Available Plugins	Ansible, Azure Resource Manager, Chef, Docker Compose, Heat Orchestration Template, Kubernetes, Terraform, Puppet, Cloudify, AWS CloudFormation, Salt, Juju, CFEngine	Ansible, Terraform
Intermediate representation	The context passed to all plugins contains a graph representing the input model	The parser generates a JSON intermediate representation which contains all the parameters for each object to be generated, including the target language and the target provider

Templates used	File names fixed in the code	Dynamically loaded from the tool's template library, based on the input model element, the target language, and the provider
Template engine	Custom, written in Java	Jinjia2

From the above, we can see that DOML+ICG focuses on more target activities including Provisioning, Configuration, Deployment and Orchestration. Both languages are based on the domain-specific metamodels and are extensible. Their code generation are based on the same strategy, i.e., Template Based Code Generation (TBCG). Although, for the moment, DOML+ICG is mainly focused on the translation to Terraform and Ansible, other IaC languages are to be supported by the same way using DOML-E mechanism. Other aspects of these two languages related to, for instance, the intermediate representation, template engine, etc., are also different and compared in detailed in the above table.

9 Plan for Future Development

The plan for future development of DOML follows both the needs of the PIACERE case studies and the requirements identified at the beginning of the project. Whenever new requirements are coming from use cases or partners, DOML is supposed to incorporate the new functionalities to support them through either direct modification on DOML or new implementation with DOML-E mechanism.

In terms of general characteristics of DOML, **REQ76** and **REQ57** are to be addressed:

- Allow the user to model information needed for each of the four considered DevOps activities (Provisioning, Configuration, Deployment, Orchestration).
- Enable both forward and backward translations from DOML to IaC and vice versa.

Regarding the specific DOML elements, **REQ28**, **REQ30**, **REQ59**, **REQ60** and **REQ36** are to be addressed:

- Support the modelling of containerized application deployment (e.g., pull, run, restart, stop docker containers).
- Enable support for policy definition constraints for QoS/NFR requirements.
- Allow users to define rules and constraints for redeployment, reconfiguration, and other mitigation actions.
- Support the modelling of security metrics both at the level of infrastructure and application.
- Enable writing infrastructure tests.

Other possible improvements involve the DOML in combination with other PIACERE components:

- Improve the representation of Application and Optimization layer. This improvement will be addressed in cooperation with the IOP development.
- Improve the DOML-extension mechanism.
- Improve the integration of Infrastructure Element Catalogue.
- Support IoT applications and infrastructures.
- Support the generation of more IaC languages. This improvement will be performed in cooperation with the development of ICG.
- Organize the DOML in multiple files.
- Implement graphical representation of DOML. This improvement will mainly concern the IDE.

10 Conclusions

This deliverable presents the new development results of DOML. The changes introduced into DOML 2.1 include the implementation of new functionalities and the solution results of fixing the problems of DOML raised during the evolution of the project. Examples of DOML models are updated to demonstrate the modelling details under the new version of DOML. The status of the requirement fulfilment is analysed, based on which, the future development plan of DOML is summarized. Besides, we discuss the differences between the DOML and the similar EDMM approach to reflect the novelty of DOML.

References

[1] E. Di Nitto ed., «Deliverable D3.1 PIACERE Abstractions, DOML and DOML-E - v1,» Piacere consortium, Dec. 2021.

[2] PIACERE team, "PIACERE DOML Specification v 2.1.," https://www.piacere-doml.deib.polimi.it/specifications/DOML_Specification_v2.1.pdf, October 2022.

[3] M. Wurster, U. Breitenbücher, M. Falkenthal and e. al, "The essential deployment metamodel: a systematic review of deployment automation technologies," *SICS Softw.-Inensiv. Cyber-Phys. Syst. 35,* p. 63–75, 2020.

[4] Morganti, Emanuele, «Deliverable 2.1 PIACERE DevSecOps Framework Requirements specification, architecture and integration strategy - v1,» Dec. 2021.

[5] Morganti, Emanuele, «Deliverable 2.2 PIACERE DevSecOps Framework Requirements specification, architecture and integration strategy - v2,» To be published.