



PIACERE

Deliverable D3.1

PIACERE Abstractions, DOML and DOML-E - v1

Editor(s):	Elisabetta Di Nitto
Responsible Partner:	Politecnico di Milano/Polimi
Status-Version:	Final – v1.0
Date:	23.12.2021
Distribution level (CO, PU):	Public

Project Number:	101000162
Project Title:	PIACERE

Title of Deliverable:	D3.1 - PIACERE Abstractions, DOML and DOML-E - v1
Due Date of Delivery to the EC	30.11.2021

Workpackage responsible for the Deliverable:	WP3 - Plan and create Infrastructure as Code
Editor(s):	Politecnico di Milano/Polimi
Contributor(s):	Go4it, HPE, Prodevelop, Tecnalía, XLAB, 7Bulls
Reviewer(s):	Alfonso de la Fuente (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 will contain the metamodel and the corresponding semantic and 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
Licensing information:	This work is licensed under Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) http://creativecommons.org/licenses/by-sa/3.0/
Disclaimer	This document reflects only the author's views and neither Agency nor the Commission are responsible for any use that may be made of the information contained therein

Document Description

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
v0.1	31.03.2021	First draft version focusing on the state of the art analysis	Polimi
v0.2	31.09.2021	Comments and suggestions received by consortium partners, contributions to the state of the art sections	ALL
v0.3	25.11.2021	Sections concerning the DOML added	Polimi
v0.4	10.12.2021	Refinements and finalization for the internal review process	Polimi, Go4It
v0.5	13.12.2021	Sent for review	ALL
v0.6	15.12.2021	Reviewed	Prodevelop
v0.9	21.12.2021	Final revision after review	ALL
V1.0	23.12.2021	Ready for submission	TECNALIA

Table of contents

Terms and abbreviations.....	6
Executive Summary.....	7
1 Introduction	9
1.1 About this deliverable	9
1.2 Document structure	10
2 State of the art analysis.....	11
2.1 Infrastructure as Code (IaC) Languages.....	11
2.1.1 Configuration management	11
2.1.2 Server provisioning and application deployment.....	13
2.1.3 Deployment, monitoring and self-adaptation	18
2.2 Modelling approaches supporting the generation of IaC.....	22
2.2.1 CAMEL	22
2.2.2 DICE Profile	24
2.2.3 SODALITE	26
2.2.4 DECIDE modeling approach.....	27
2.2.5 RADON.....	29
2.3 Generating IaC code from a model	31
2.4 Summary	33
3 Requirements for the DOML.....	34
4 DOML modelling principles	38
4.1 A single model for multiple IaC code fragments	38
4.2 Separation of concerns and multiple modelling layers	39
5 Modelling abstractions relevant for IaC definition	41
5.1 Adopted approach.....	41
5.2 Common elements	42
5.3 Application layer	43
5.4 Abstract infrastructure layer	43
5.5 Concrete infrastructure layer	44
5.6 Optimization.....	45
6 The DOML v0.1 language	47
7 First DOML examples	48
7.1 WordPress Website.....	48
7.1.1 DOML model with a VM database.....	48
7.1.2 DOML model with a SaaS database.....	53
7.2 FaaS Thumbnail Generator.....	55
7.2.1 Application Layer	56

7.2.2	Abstract Infrastructure Layer	56
7.2.3	Concrete Infrastructure Layer	57
8	Reflections on the development of the extension mechanism DOML-E	59
9	Conclusions	61
10	References.....	62

List of tables

TABLE 1. COMPARISON OF AUTOMATION TOOLS FOR DEVOPS [3], [4]	12
TABLE 2. CLASSIFICATION OF TBCG TOOLS [43]	32
TABLE 3. REQUIREMENTS ON THE GENERAL CHARACTERISTICS OF THE DOML [52].	34
TABLE 4. REQUIREMENTS ON THE SPECIFIC ELEMENTS TO BE MODELLED IN THE DOML.....	35

List of figures

FIGURE 1. TYPICAL OPS ACTIVITIES (ADOPTED FROM [2])	11
FIGURE 2. STRUCTURAL ELEMENTS OF A TOSCA TEMPLATE AND THEIR RELATIONSHIPS [10]	15
FIGURE 3. THE TERRAFORM WORKFLOW (ADAPTED FROM [13]).....	17
FIGURE 4. CLOUDFORMATION WORKFLOW [14]	17
FIGURE 5. SOFTWARE LIFECYCLE BEFORE AND AFTER DOCKER	19
FIGURE 6. CAMEL MAIN CONCEPTS	23
FIGURE 7. DATA INTENSIVE APPLICATIONS CONCEPTS IN DICE FROM [27].	24
FIGURE 8. SODALITE HIGH-LEVEL ARCHITECTURE OF THE DESIGN-TIME ENVIRONMENT [29].	26
FIGURE 9. DECIDE INTEGRATED GENERIC ARCHITECTURE [31].	28
FIGURE 10. EXCERPT OF AN APPLICATION DESCRIPTION FILE.	29
FIGURE 11. RADON ARCHITECTURE [34].	30
FIGURE 12. RELATIONSHIPS BETWEEN A COMPONENT AND THE EXECUTION ENVIRONMENT IT RUNS ON	38
FIGURE 13. MODELLING THE APPLICATION STRUCTURE	39
FIGURE 14. MODELLING THE INFRASTRUCTURE AND THE MAPPING WITH COMPONENTS	39
FIGURE 15. MODELLING AN ABSTRACT INFRASTRUCTURE AND THE MAPPING WITH COMPONENTS	40
FIGURE 16. MODELLING DIFFERENT CONCRETIZATIONS OF AN ABSTRACT INFRASTRUCTURE	40
FIGURE 17. STRUCTURE OF THE COMMON ELEMENTS LAYER.....	42
FIGURE 18. STRUCTURE OF THE APPLICATION LAYER	43
FIGURE 19. STRUCTURE OF THE ABSTRACT INFRASTRUCTURE LAYER	44
FIGURE 20. STRUCTURE OF THE CONCRETE INFRASTRUCTURE LAYER.....	45
FIGURE 21. STRUCTURE OF THE OPTIMIZATION LAYER.....	46
FIGURE 22. DIAGRAM OF THE WORDPRESS EXAMPLE WITH A VM DATABASE	48
FIGURE 23. DIAGRAM OF THE WORDPRESS EXAMPLE WITH A SAAS DATABASE.....	54
FIGURE 24. DIAGRAM OF THE EXAMPLE FAAS THUMBNAIL GENERATOR.	55

Terms and abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
CAMEL	Cloud Application Modelling and Execution Language
CERIF	Common European Research Information Format
CMS	Content Management System
CNI	Container Networking Interface
CRD	Custom Resource Definition
CSI	Container Storage Interface
CSP	Cloud Service Provider
DBMS	DataBase Management System
DevOps	Development and Operations
DoA	Description of Action
DOML	DevOps Modeling Language
DOML-E	DOML Extension Mechanism
DSL	Domain-Specific Language
EC	European Commission
EMF	Eclipse Modeling Framework
FaaS	Function as a Service
GA	Grant Agreement to the project
GCP	Google Cloud Platform
IaC	Infrastructure as Code
ICG	Infrastructural Code Generator
IEP	IaC Execution Platform
IOP	IaC Optimization
KPI	Key Performance Indicator
OCL	Object Constraint Language (part of EMF)
REST	REpresentational State Transfer
SaaS	Software as a Service
SRL	Scalability-Rule Language
SW	Software
TOSCA	Topology and Orchestration Specification for Cloud Applications
VM	Virtual Machine
VT	Verification Tool

Executive Summary

The main objective of the PIACERE project is to provide means (tools, methods and techniques) to enable most organizations to fully embrace the Infrastructure-as-Code approach, through the DevSecOps philosophy, by making the creation of such infrastructural code more accessible to designers, developers and operators (DevSecOps teams), increasing the quality, security, trustworthiness and evolvability of infrastructural code while ensuring its business continuity by providing self-healing mechanisms anticipation of failures and violations, allowing it to self-learn from the conditions that triggered such re-adaptations.

In particular, PIACERE aims at allowing DevSecOps teams to model different infrastructure environments, by means of abstractions, through a *DevOps Modelling Language* (DOML) that hides the specificities and technicalities of the current solutions and increases the productivity of these teams (DOML is called Key Result 1 in PIACERE). Models defined in the DOML are then translated, through the *Infrastructural Code Generator* (ICG – PIACERE Key Result 3), into the target languages needed by the existing IaC tools, to reduce the time needed for creating infrastructural code for complex applications.

Another important issue to consider is that in the current highly dynamic and evolving context of new computing resources as well as new IaC languages and tools is continuously developing. This requires the definition of proper extensibility mechanisms for the DOML and the corresponding ICGs to ensure the sustainability and longevity of the PIACERE approach and tool-suite. This is the purpose of the *DOML Extension mechanisms* (DOML-E – PIACERE Key Result 4). These will allow new infrastructural components and IaC tools to be incorporated in the DOML language, be they for software execution, network communication, cloud services, or data storage.

DOML models are created through the PIACERE IDE (PIACERE Key Result 2), which supports users in their activities through suggestions and guidance and integrates all other design-time PIACERE tools.

The main goal of this deliverable is to define the first version of the PIACERE DOML and to show, through some examples, how it can be used in practice. The DOML is defined as a set of concepts to be modelled in order to properly represent application components and the resources they should rely on, plus a syntax which is offered to the end users as a tool to define a model.

Such first version of the modelling approach comes out from a thorough analysis phase. This included a careful analysis of the literature concerning IaC languages as well as other modelling approaches and a critical review of the requirements for the DOML language. These requirements have been collected through multiple discussions with the owners of PIACERE case studies as well as with the other technical partners focusing on the other aspects of the project (the verification, the optimization of the deployment configuration, the canary environment, the coordination of the orchestration, the support to operation and evolution).

A summary of the outcomes of these activities is reported in this deliverable as well, together with the definition of the main design principles behind the DOML, that is:

- 1) the possibility to define a single model from where IaC in different languages and for different phases can be created
- 2) the principle of separation of concerns which has led us to organize the language into multiple layers, each focusing on the modelling of a specific aspect.

The extension mechanism (DOML-E) will be analysed in detail in the next iteration of the work. However, the DOML itself has been developed keeping in mind the need for extensibility and,

therefore, some preliminary decisions in the direction of supporting DOML-E have been taken and are briefly reported as well.

This deliverable is an integral part of Work Package 3 within the PIACERE project. Its content, namely, the definition of the first version of the DOML and the definition of the main characteristics of DOML-E are related to the ICG presented in Deliverable 3.4 and the IDE, presented in Deliverable D3.7. As the DOML is verified through the Verification Tool (VT), there is also an important dependency with the corresponding Deliverable D4.1. Since the work in all these areas has been conducted in parallel during this first year of the project, the DOML representation adopted in each of these deliverables is not yet fully aligned as we have been experimenting with multiple types of syntax and various approaches. However, the main concepts associated to a DOML specification are shared and common and are those described in this deliverable.

Coherently with its objectives, this deliverable begins with a section concerning preliminaries where an analysis of the state of the art is developed. The deliverable then follows with an analysis of the DOML requirements including how the chosen approach is addressing them. The core of the deliverable includes the definition of the design principles behind the DOML, the presentation of the defined modelling abstractions, the development of the language and a set of examples that show how it can be used in practice. The last part of the deliverable progresses toward the end with a preliminary discussion on the DOML-E followed with the conclusion.

The deliverable is accompanied by an Annex that provides a detailed definition of all concepts that are part of the DOML and that will be released as a separate document to facilitate its usage and evolution independently of this deliverable.

The plan for the next iteration of the DOML includes its validation through the PIACERE use cases and, possibly, through other examples. The experiments will allow us to check the completeness of the language and its ability to provide to the ICGs all pieces of information needed for the translation into the target IaCs. Moreover, they will allow us to check the verifiability of the DOML using the model checking-based approach offered by the VT (Verification Tool) and the expressiveness of the language for what concerns the optimization features. Based on the outcomes of these experiments, a new more polished version of the language will be created. Particular attention will be posed to the DOML-E and to the ability to incorporate in the DOML all aspects to be modelled in order to make all PIACERE tools as flexible as possible. The results on these activities will be reported in D3.2 at Month 24 and D3.3 and Month 30.

1 Introduction

laC (Infrastructure as Code) has introduced the possibility to program beforehand the way software is deployed and configured on some execution environment composed of Virtual Machines and/or various kinds of containers. Thanks to this laC programming effort, it is possible to replicate a deployment multiple times by simply running a script, to keep the characteristics of the operational environment under control, and to speed up the time to market for a certain product.

However, building laC is not a trivial task. It requires an in-depth knowledge of both the laC language to be used and the characteristics of the target operational environment. Moreover, laC approaches still suffer of the following issues:

- i) There is a large variety of competing tools requiring the adoption of different programming languages for writing infrastructural code.
- ii) All these tools and languages are focusing on a single or a small set of automation steps and of types of resources (e.g. VMs).
- iii) They mostly focus on cloud computing leaving aside other computational resources such as the edge.
- iv) They focus on certain phases of the lifecycle of the laC such as provisioning, configuration or deployment but there is not really an end-to-end solution covering the Devs and the Ops. As an example, one can use the TOSCA language to describe the VMs to be provisioned and the software layers to be deployed on top of them. However, the specific actions to be executed on such resources to configure software components will have to be coded in a different language such as Ansible.

In this context, PIACERE aims at allowing DevSecOps teams to model different infrastructure environments, by means of abstractions, through a *DevOps Modelling Language* (DOML) that hides the specificities and technicalities of the current solutions and increases the productivity of these teams. Models defined in the DOML are then translated, through the *Infrastructural Code Generator* (ICG), into the target languages needed by the existing laC tools, to reduce the time needed for creating infrastructural code for complex applications.

DOML models are created through the PIACERE IDE, which supports users in their activities through suggestions and guidance and integrates all other design-time PIACERE tools.

Another important issue to consider is that in the current highly dynamic and evolving context new computing resources as well as new laC languages and tools are continuously emerging. This requires the definition of proper extensibility mechanisms for the DOML and the corresponding ICGs to ensure the sustainability and longevity of the PIACERE approach and tool-suite. This is the purpose of the *DOML Extension mechanisms (DOML-E)*. These will allow new infrastructural components and laC tools to be incorporated in the DOML language, be they for software execution, network communication, cloud services, or data storage.

1.1 About this deliverable

The main goal of this deliverable is to define the first version of the PIACERE DOML and to show, through some examples, how it can be used in practice. As we will discuss through the deliverable, the DOML is defined as a set of concepts to be modelled in order to properly represent application components and the resources they should rely plus a syntax which is offered to the end users as a tool to define a model.

Such first version of the modelling approach comes out from a thorough analysis phase where the literature concerning laC languages as well as other modelling approaches has been carefully

and critically studied and the requirements for the DOML language have been collected through multiple discussions with the owners of PIACERE case studies as well as with the other technical partners focusing on the other aspects of the project (the verification, the optimization of the deployment configuration, the canary environment, the coordination of the orchestration, the support to operation and evolution). A summary of the outcomes of these activities is reported in this deliverable as well.

The extension mechanism (DOML-E) will be analysed in detail in the next iteration of the work. However, the DOML itself has been developed keeping in mind the need for extensibility and, therefore, some preliminary decisions in the direction of supporting DOML-E have been taken and are briefly reported as well.

1.2 Document structure

The rest of this document is structured as follows:

- Section 2 focuses on the analysis of the state of the art for what concerns the following aspects: Infrastructure as Code (IaC) languages, modelling approaches supporting the generation of IaC, code generation approaches.
- Section 3 presents the requirements for the DOML as they have been collected in the interaction with the PIACERE partners.
- Section 4 presents the main modelling principles behind the DOML
- Section 5 presents the abstractions that are relevant to modelling the aspects of a software application relevant in one of the phases of the Ops cycle.
- Section 6 presents the general structure of the DOML language first version and defines its main characteristics, based on the findings of the previous sections.
- Section 7 presents two examples of use of the DOML.
- Section 8 highlights the preliminary ideas on how to create the extension mechanism DOML-E.
- Finally, Section 9 defines the plan for the evolution of the presented results and draws the conclusions.

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

2 State of the art analysis

This section analyses the state-of-the-art of Infrastructure as code (IaC) languages and different modelling approaches supporting the generation of IaC, and also the generation techniques (e.g. the template-based code generation). The IaC languages considered include TOSCA, Terraform, Ansible, Kubernetes, Docker, Chef, Puppet and CloudFormation. The modelling approaches include CAMEL / Melodic / Morphemic, DICE, SODALITE, DECIDE and RADON.

2.1 Infrastructure as Code (IaC) Languages

Figure 1 from [2] shows the typical Ops activities required to continuously deploy and operate Cloud applications:

- *Configuration management* is the process of deploying and managing at runtime all the required services (e.g., Tomcat, MySQL, Hadoop, Cassandra). It consumes reusable recipes that (often declaratively) describe how to manage and configure the service across lifecycle phases.
- *Server provisioning* entails acquiring (e.g. from a public provider of Infrastructure-as-a-Service), configuring and running the required VMs or containers upon which services can run.
- *Application deployment* is the phase in which user's applications are executed on the resulting infrastructure.
- *Monitoring* concerns all the running component (VMs, services, middleware and applications) and is a basic enabler for almost any kind of runtime management activity.
- *Self-adaptation* is about applying a set of methods (e.g. VMs autoscaling, faults recovery, etc) to keep the status conforming with service-level agreements and the quality of service (QoS) variables therein (e.g., security, network safety, etc.).

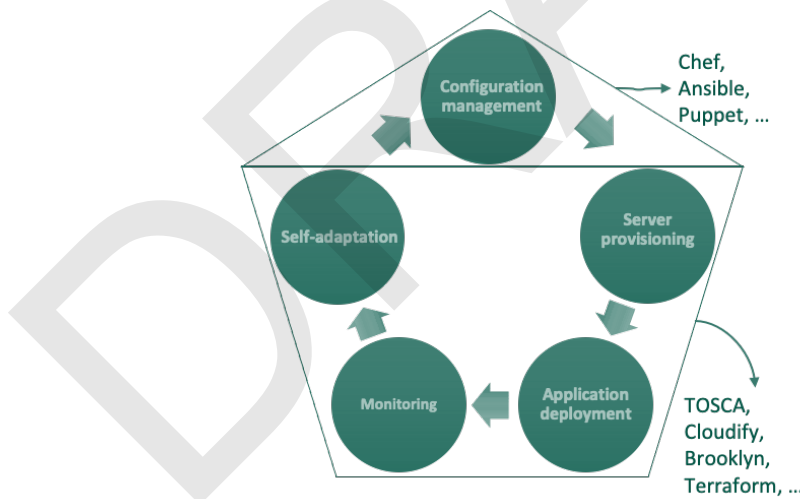


Figure 1. Typical Ops Activities (adopted from [2])

All these activities require specific languages and tools. In this section we provide an overview of the most well-known approaches based on the usage of some kind of IaC, that cover the various phases described above.

2.1.1 Configuration management

The most well-known approaches for configuration management are Chef, Puppet and Ansible (see Table 1). Each of them offers a specific IaC language. In the following of this section we provide a brief overview of these three approaches.

Table 1. Comparison of automation tools for DevOps [3], [4]

Tool	DevOps phase	Tool type	Configuration format	Language	License
Puppet	Deployment	Configuration management	DSK similar to JSON (JavaScript Object Notation)	Ruby	Apache
Chef	Deployment	Configuration management	Ruby-based DSL	Ruby	Apache
Ansible	Deployment	Configuration management	YAML (YAML Ain't Markup Language)	Python	GPL (GNU General Public License)

2.1.1.1 Chef

Chef [5] is one of the supporters of the DevOps software lifecycle and of the Infrastructure as Code approach. It allows automation of application deployment and configuration on multiple resources from edge devices to on-premises or on-cloud servers.

Chef includes an enterprise dashboard, Automate, that simplifies the execution of deployment and configuration tasks. Automate is supported by some open source tools, the most important ones being Chef Infra, which is in charge of automating the configuration, deployment and management of an infrastructure, and Habitat, which defines what should happen when a new application version is created or when a runtime failure is discovered and the system must rollback into the previous version.

The interesting aspect of Chef Infra is its distributed architecture. Each node of a certain infrastructure is supposed to run a Chef client. This client is periodically contacting the Chef server to check whether the node configuration is aligned with the one defined in the server. If there is a misalignment, for instance, because a user has decided to change the configuration in the server or because the configuration of the node itself has changed because of some failure, the client will execute all steps needed to realign the node to the server configuration. Configurations are defined thanks to the Chef Language in terms of recipes and cookbooks. These can be reused in multiple contexts and are typically made available to the community. Users interact with Chef Infra through the Chef Workstation. This includes a IaC development environment featured with a testing environment (Test Kitchen).

2.1.1.2 Puppet

Puppet [4] is an open-source declarative configuration management tool used to manage and automate the configuration of servers. When using Puppet, one defines the desired state of the systems in the given infrastructure to be managed. This is done by writing infrastructure code in the Puppet's Domain-Specific Language (DSL) that is a Ruby-like language. Puppet code works for a wide range of devices and OS.

Puppet, like Chef, has been designed as an agent-based solution and, thus, it requires an agent to be installed on each infrastructure node and a primary server that is coordinating the work of the various agents. The primary server stores the code written in the Puppet DSL. The Puppet agent executes the code associated to the node where it is running interacting with the operating system of that node. This is called a Puppet run [6]. Agents then inform the primary server about the state of their nodes so that the server has a complete overview of the state of the whole system and can check whether it is the desired one or whether other actions need to be executed. This ensures that the configuration is kept consistent. Another advantage offered by Puppet stems in the possibility to automate the configuration of a heterogeneous

infrastructure and to apply the same Puppet code to a large number of servers that can increase overtime, thus saving costs and effort.

2.1.1.3 Ansible

Red Hat Ansible [7] is an open-source IT automation tool used for automating the configuration management of devices and services. Ansible enables users to automate almost anything from straightforward application deployment to complex cloud infrastructure management. This Python-based tool is often used for configuring the remote virtual machines by installing necessary software via portable YAML scripts called Ansible playbooks. In other words, Ansible can be seen as a powerful replacement for shell scripts with the ability to facilitate running multiple sequential tasks on multiple clients at the same time. These tasks run solely on the remote machines and not locally. Apart from being serverless and client-based, an important advantage is that the developer can specify the desired state of the target machine regardless of the running process. To achieve this, Ansible uses an inherently simple agentless approach to remote infrastructure management, which is implemented through the standard Python Paramiko SSH library enabling the DevOps to manage any infrastructure accessible through SSH.

Ansible introduces six main IaC components, which are:

- inventories,
- modules,
- roles,
- playbooks,
- roles, and
- plugins

Inventory includes the list and properties of target machines that will be managed. Ansible includes certified collections and community modules that are used to automate different tasks. These tasks are organized and specified in detail within Ansible playbooks that represent the main entity of running. On the other hand, Ansible roles already include multiple groups of tasks with necessary parameters prepared for execution on the target machine. With Ansible plugins, users can extend the elemental functionalities and define a new flexible set of functions. All these aforementioned components, accompanied by tests and documentation, can be packed and published as Ansible collections (for example collections for AWS, Azure, GCP, etc.).

Ansible, like Terraform, is an industrial Infrastructure as Code (IaC) tool. It offers a simple and clean declarative IaC language which is widely accepted and easy to learn and adopt. Besides automation, the tool can be also used as an orchestrator, where Ansible roles are used to order and group together multiple tasks to achieve a coherent deployment entity. Ansible is characterized by a vast support community and probably the largest set of cloud infrastructure libraries support (i.e., Ansible Galaxy). The toolbox also includes an enterprise Red Hat Ansible Tower that facilitates the orchestration of complex tasks through GUI.

2.1.2 Server provisioning and application deployment

Under this category fall various approaches. Among the others, we mention TOSCA, which is a standardization initiative by OASIS, Terraform, which is gaining momentum and shows an increasing number of users and interest by practitioners, and CloudFormation, which is an AWS initiative specifically tailored for their cloud.

2.1.2.1 TOSCA

TOSCA [8] stands for Topology and Orchestration Specification for Cloud Applications and is governed by the OASIS standard. It focuses on defining a container orchestration specification that is portable across various cloud environments and container providers.

“The goal of TOSCA is to try to resolve the problem enhancing the portability of Cloud applications and the IT services that comprise them, running on complex software and hardware infrastructure (2014).”

TOSCA is a standard IaC language that is designed to support a Cloud information model that can be extended through the definition of new node types and through inheritance. When the deployment model is available, then an orchestrator can execute it and deploy the corresponding components on the available resources. Furthermore, it is defined as implementation-agnostic.

TOSCA is a declarative language. There is no vendor lock-in, and it covers components, relationships, policies, workflows and processes of Cloud services. Its concepts can be applied to practically any infrastructure. An execution engine (orchestrator) interprets the topology and creates states.

The TOSCA format is YAML compliant, it includes definitions of lifecycles, relationships, policies, and plans that also describe the operational aspect of the application services. TOSCA defines only the topology, but not how such topology will be executed. The executors can be Ansible, Terraform, Chef, Puppet, Bash, etc. Every orchestrator understands the TOSCA part but might not be able to deploy a TOSCA specification due to the execution specifics.

Figure 2 from [9] presents the main concepts offered by TOSCA. A node type can be specialized to represent various types of elements, from application-level components to virtual machines or other execution environments, middleware elements and the like. A relationship type defines the characteristics of a relationship that can hold between nodes of certain types. Through such relationship types, for instance, we can model the fact that a certain type of application component should be executed on a certain type of VM.

Templates are instantiations of specific types. In particular, a topology template may describe the configuration of a specific deployment.

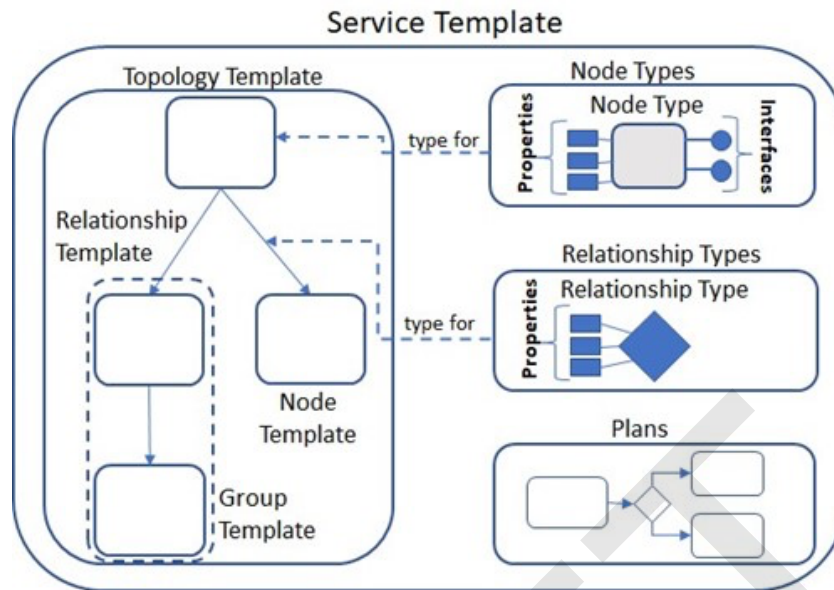


Figure 2. Structural elements of a TOSCA template and their relationships [10]

TOSCA has different features:

- We can separate the topology definition from the lifecycle operation definition. It supports both hardware and software components which means that we can define hardware and software resources and relationships with services.
- We can define capability requirements between nodes and the way to instantiate these nodes can be implemented in separate languages.
- We can define relationships between hardware and software resources and those services, and it is very important to have a high-level definition of the model, comprising all aspects of the model.
- Nodes can expose both capabilities and requirements. These can be matched with each other by an orchestrator.
- TOSCA also enables the modelling of node inheritance, which means that we can derive nodes from other nodes. The user can also overwrite some implementation operations. In a way it is quite like other object-oriented programming languages, which is really important because it gives an additional layer of abstraction, and an additional layer of reusability of the nodes' definition.
- It also features relationship definition and configuration, which means that a relationship can connect nodes and perform additional steps for configuration of the nodes.
- It also can model Service Level Agreement (SLA, or policies) through the definition of event triggering and similar policy definitions.
- Ultimately, Tosca has the potential for reducing the cost of developing new, or improving existing, programming languages, either general-purpose or domain-specific.

The specification for orchestration is well matured, TOSCA orchestration progresses with a good speed of development and has the potential for becoming the standard blueprint definition for containers. It has a standard orchestration format (for example, it is simple to orchestrate Docker workloads using standard TOSCA YAML) and it covers the entire application lifecycle, including post-deployment aspects such as monitoring, additional workflows (continuous deployment, scaling out), and policies to automatically trigger some of these workflows.

Associated to the language, there are multiple implementations, both commercial and open-source, such as: Juju, Cloudify, IBM Cloud Orchestrator, OpenStack Heat. It has also been adopted by Telco vendors such as Alcatel-Lucent, Huawei, and Cisco.

Within the PIACERE project a possible orchestrator for TOSCA could be xOpera [11] which is a lightweight open source, state-aware orchestrator.

The xOpera project combines TOSCA standard with the most popular declarative tool Ansible. Exploiting the combination, DevOps developers can use human-readable deployment blueprints to create deployment graphs and relationships that can overpower even the most prominent orchestrators currently available.

Definitely, TOSCA allows users to specify any type of IT system. However, this generality results into a language which is quite verbose and difficult to be used. For this reason, other projects, such as SODALITE (see Section 2.2.3) build editors and abstraction layers that aim at guiding the users in the adoption of TOSCA.

2.1.2.2 Terraform

Terraform [12] is an Infrastructure-as-code (IaC) tool for building, changing, and versioning infrastructure, written in Go and created by a company called HashiCorp (originally launched in 2014).

It is compatible with many different providers, from the major ones such as Amazon AWS, Azure, and Google Cloud Platform (GCP), and it can also manage custom in-house solutions. Terraform is able to manage both low-level components such as virtual machines, storage, and networks, and high-level components such as DB as a Service.

It is one of the most used IaC tools, used by DevOps teams to automate infrastructure-related tasks and it has a very large community and a lot of open-source modules. The main reason to choose Terraform is perhaps that it is an open-source provisioning tool supporting multiple cloud providers, offering an agentless architecture, a declarative language, a large community, and a mature codebase.

Terraform configuration files (the IaC code) are written in HCL (Hashi Corp Configuration Language) and define the desired state of the system. This makes Terraform a declarative infrastructure tool. Terraform keeps track of the current state of the infrastructure and, when it applies a new configuration, compares it with the stored state to understand what needs to be changed. This is different from many other (imperative) tools, which describe every step specifically. The HCL language is designed to be both human- and machine-friendly.

Features of Terraform

- **Infrastructure as a Code:** Configuration files can be versioned and treated as any other source code. In this way, repeatability is improved, and the infrastructure can be shared and reused.
- **Execution Plans:** Terraform workflow has a “planning” step that generates an execution plan. This execution plan details the actions that Terraform will execute when the plan is applied. This increases confidence and allows verifying if the actions are as intended, before manipulating the real infrastructure.
- **Resource Graph:** Terraform users can get information about dependencies in their infrastructure in terms of a resource graph.
- **Change Automation:** By executing Terraform configuration files, complex infrastructure changes can be applied in an automated way.

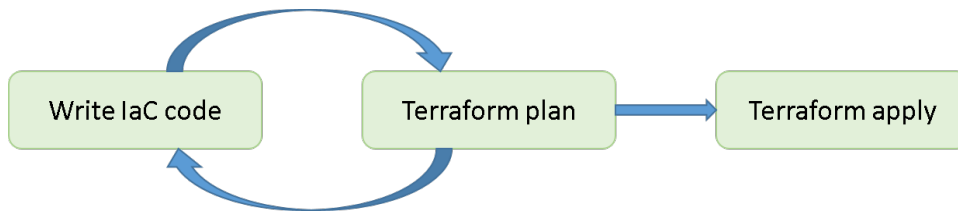


Figure 3. The Terraform workflow (adapted from [13])

2.1.2.3 CloudFormation

CloudFormation [14] is the Amazon Web Services (AWS) *Infrastructure as Code (IaC)* tool, and offers a manner for interacting with the AWS stack following the IaC philosophy.

Figure 4 **Error! Reference source not found.**from [14] depicts the basic, high-level functioning of AWS CloudFormation:

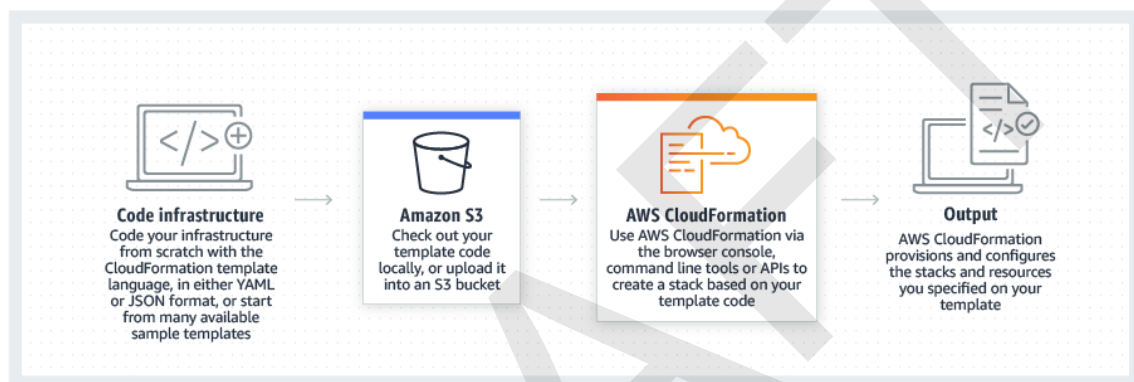


Figure 4. CloudFormation workflow [14]

AWS CloudFormation is a set of text files that map to the different modules and tools available in the AWS stack. This way, the different services (e.g., S3 for storage, EC2 for computational power, Lambda for functions) can be modelled and the benefits of having them in this format are automatically obtained by the project in hand. These are: versioning capabilities, reproducibility, reusability, portability. Given that CloudFormation scripts are in essence text files, they can be incorporated into the CI/CD pipelines of the project, and at the same time can evolve with it accordingly. Scripts developed with CloudFormation are deterministic and their functionality can be reproduced between different projects. At the same time, CloudFormation scripts are portable, and their functionality is stable no matter who executes them.

CloudFormation provides a manner of defining templates. A CloudFormation template describes the desired resources and their dependencies so that they can be launched and configured together as a stack. A template can be used to create, update, and delete an entire stack as a single unit, as often as it is needed, instead of managing resources individually [15].

With CloudFormation, DevOps and GitOps best practices can be applied by using widely adopted processes such as starting with a git repository and deploying through a CI/CD pipeline [16]. Auditing changes can also be simplified, and it is possible to trigger automated deployments with pipeline integrations such as GitHub Actions and AWS CodePipeline.

Moreover, it can help to manage resource scaling by sharing CloudFormation templates to be used across an organization, to meet safety, compliance, and configuration standards across all AWS accounts and regions. Templates and parameters enable easy scaling so that best practices and company policies can be shared. Additionally, CloudFormation StackSets [14] enables the

possibility to manage and provision (create, update, or delete) stacks across multiple AWS Accounts and Regions, with a single operation.

Naturally, to further automate resource management across an organization, CloudFormation can be integrated with other AWS services, including AWS Identity and Access Management (IAM) for access control, AWS Config for compliance, and AWS Service Catalog for turnkey application distribution, and additional governance controls. Integrations with CodePipeline and other builder tools allow to implement the latest DevOps best practices and improve automation, testing, and controls.

It is also possible to model, provision, and manage third-party application resources (such as monitoring, team productivity, incident management, CI/CD, and version control applications) alongside any AWS resources, and to use the open source CloudFormation CLI (Command Line Interface) to build entirely owned CloudFormation resource providers – native AWS types published as open source.

The AWS CloudFormation GitHub code repository [14] offers open source projects that extend CloudFormation’s capabilities. The CloudFormation Registry and CloudFormation CLI permit the definition and creation of resource providers to automate the creation of resources safely and systematically. Using CloudFormation GitHub projects, CloudFormation templates can be checked for policy compliance or validate use of best practices.

2.1.3 Deployment, monitoring and self-adaptation

The activities related to the execution control of software systems include PaaS and IaaS services as well as specific frameworks that, thanks to the creation of proper abstraction layers, can manage the mapping between software components and computational resources in a simple and seamless way. In this section we focus on the two most well-known approaches that belong to this last category, namely, Docker and Kubernetes.

2.1.3.1 Docker/Docker Swarm

Docker [17, 18] has become the standard the facto for running container based applications on-premises, and in public and private cloud providers. However, it is not the only solution as there exist other containerization technologies that can be used such as LXC [19]. It has a public GitHub with seventy different repositories that offer a large variety of functionalities in the docker ecosystem (e.g., compose, kitematic, labs).

The large adoption of docker has been due to its ability for offering lightweight virtual environments that are portable, self-contained, and multiplatform. Firstly, it solves the well-known problem of portability because a containerized application can be executed on different platforms and not only in the developer’s laptop, hence avoiding last minute issues in the deployment stage. Secondly, it offers the ability of packaging the application with all the necessary dependencies in their correct version, avoiding this way problems with libraries being previously installed (i.e., backward portability issues). In addition, the application is shipped with the operative system itself in the proper version approved by the organization. Thirdly, it is multiplatform as an application being shipped with the docker technology should be able to run in a large variety of servers regardless of their operative system and version. A lot of efforts have been done towards this goal and the biggest operative systems support docker natively (e.g. Windows, MacOS, Linux).

The docker platform promotes the use of continuous integration and deployment techniques as it can be easily integrated in the development flow. First, the entire specification of the docker

container can be defined in a text file, which is very suitable for being integrated in source code repository and enables a correct versioning of the applications. Then, platforms like Gitlab offer built-in CI/CD applications which enable the application to be self-shipped and self-contained, and that every stage of the development cycle (i.e., compile, package, test, analyze, deploy, ...) can be executed in a centralized way.

Due to this, docker has paved the way for the development and operations teams to work closer together and this closer collaboration promotes the good practices that raise the success rate of development projects. Docker has changed all of this, allowing different professionals involved in this process to effectively speak one language, easing collaboration. Everything goes through a common pipeline to a single output that can be used on any target so there is no need to continue maintaining a bewildering array of tool configurations, as shown in Figure 5. **Error! Reference source not found..**

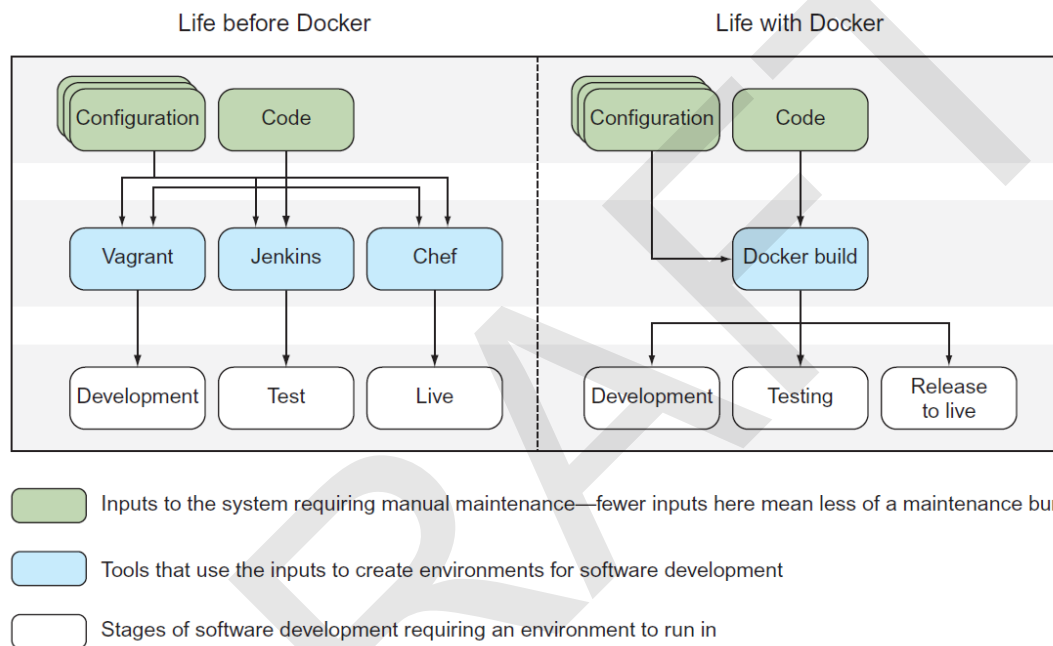


Figure 5. Software lifecycle before and after Docker

At the same time, there's no need to throw away an existing software stack if it works for its intended purpose—it can be packaged up in a Docker container as-is, for others to consume. As a bonus, it is possible to see how these containers were built, making it possible to dig into the details, if necessary.

Docker Swarm is docker's approach for providing a solution for the deployment of containers in a cluster of computers. These clusters are comprised of a minimum of three nodes and can span to thousands of them. Docker swarm provides a centralized management of the containers that live in a certain cluster. In particular, it is able to deploy, monitor, rebalance these containers and to offer certain functionalities like connectivity among the containers, and from outside the cluster. It is regarded to be an easy to use solution, stable enough to be used production environments. However, it has not been able to be the go-to solution for running containers in a distributed fashion, and other players in the market have managed to obtain bigger adoption. In this mode, nodes can obtain the role of manager or worker, the former is in charge of handling the various nuances of a cluster based deployment, whereas the latter is in charge of deploying the containers themselves.

2.1.3.2 Kubernetes

Kubernetes (usually abbreviated as K8s) *is a portable, extensible, and open-source platform for the management of containerized workloads and services, capable of facilitating both declarative configuration and automation* [20]. Google opened the Kubernetes project in 2014.

Containers are similar to virtual machines, but have a lighter isolation model, sharing the operating system (OS) kernel between applications. Therefore, containers are considered lighter. Like a virtual machine, a container can have a segregation of filesystem, CPU, memory, PID, and more. Because containers are meant to be decoupled from the underlying infrastructure, they are often designed to be portable between different clouds and different deployments. Proper approach to containers offers multiple benefits, for example:

- Separation of Dev-Ops phases: container images are produced at the time of compiling the application and not during the release phase, thus allowing the applications to be decoupled from the infrastructure.
- Environment consistency between development, testing and production.
- Portability between different cloud service providers and operating systems.
- Application-centric management.
- Resource isolation.

Kubernetes provides a framework to make distributed systems work resiliently and takes care of scalability, availability, failover, and application distribution. In addition, Kubernetes can easily manage releases with a desired deployment approach, e.g., allowing two (or more) versions of an application to coexist and enabling the operator to switch from one version to another on demand or to dynamically route traffic to multiple versions (approaches known as canary deployments, blue/green deployments and A/B testing).. Kubernetes provides e.g. [20]:

- Discovery of services and load balancing: Kubernetes can expose a container using a DNS name or its IP address, possibly distributing traffic across multiple containers so that the service remains stable.
- Storage orchestration: Kubernetes allows to automatically mount a storage system . Some of the supported options include local storage, disks provided by public clouds, and more.
- Automated rollout and rollback: the user can describe the desired state of the containers and Kubernetes can then modify the active current state to achieve the desired one.
- Load Optimization: Kubernetes can be configured to allocate containers on nodes and cluster of nodes in order to maximize the use of available resources. Configuration needs such as CPU and memory (RAM) for each container can be instructed.
- Self-healing: when a container crashes, Kubernetes can restart it whereas a container does not respond to health checks, Kubernetes can terminate it. The objective is to avoid getting traffic to containers that are not up or responding correctly.
- Management of sensitive information and configuration [21]: Kubernetes supports the storage and management of sensitive information, such as SSH keys, OAuth tokens and passwords. Sensitive information and application configuration can be deployed and updated without having to rebuild container images and without revealing sensitive information.

The main concept of working with Kubernetes is that it is declarative and REST-API-driven. The declarative approach allows Kubernetes to operate in control loop fashion by working towards ensuring that the actual state matches the desired state as stored in Kubernetes's internal database (usually etcd). The REST API is served over HTTP(S).

Working with Kubernetes revolves around manipulating objects (also known as resources, however, this other name is used only in certain contexts as it conflicts with the homonymous concept related to managed containers) which describe the desired state. These objects are serialised into the internal database, and usually represented using YAML when interacted with by humans. Each object belongs to a certain kind as supported by the target Kubernetes cluster. The object's kind defines the properties it can hold and determines its behaviour (or rather: the expected behaviour, the actual behaviour is up to the governing controller). It also decides whether the object is namespaced or cluster-wide. Namespace is also a kind of object in Kubernetes but specific in that it offers a grouping of other objects. Each namespaced object belongs to exactly one namespace. Each object also has a name which is unique in its namespace if it is namespaced, or unique cluster-wide if not.

Workloads on Kubernetes are organised using pods. Pod is the smallest unit of scheduling in Kubernetes. A pod encapsulates one or more containers which are then collocated and share certain resources. Each pod has its own unique IP address and can be addressed as such.

Kubernetes cluster architecture

A Kubernetes cluster is made up of two macro blocks, the first one called control plane and the second one called workers or, more often, simply nodes. The two blocks may be collocated on the same machines.

The control plane is the core of the cluster and runs the following components [22]:

- **kube-apiserver** - is the component that exposes the Kubernetes API and is the main component since Kubernetes has been designed and built to base all the operations on the use of the REST API.
- **etcd** - is a distributed key-value store that acts as a database that maintains all information relating to the desired and actual state of the cluster and its objects. Kube-apiserver is the only component interacting with etcd directly.
- **kube-scheduler** - schedules pods on workers, based on, e.g., the required resources, node status, affinity and anti-affinity rules, and tolerations.
- **kube-controller-manager** – the workhorse of Kubernetes, it is implemented as a set of control processes (**controllers**) which run the control loops ensuring that the actual state moves towards desired state. They are responsible, for example, for:
 - Checking if cluster nodes are active.
 - Checking if the number and status of running pods are correct.
- **cloud-controller-manager** (optional) - an optional set of controllers that interact with the underlying cloud resource provider.

The nodes are where the workload is carried out, i.e., where the pods are put into execution, and they run the following components [22]:

- **kubelet** - as kube-apiserver was the heart of the control plane, the kubelet is the heart of each node. It also offers a REST API but it is seldom called directly by Kubernetes users. Instead, they are proxied by kube-apiserver. The kubelet registers the node in the kube-apiserver and is responsible for managing pods running on that node (indirectly via the container runtime).
- **kube-proxy** - is a helper proxy service that facilitates Kubernetes support of services (as in objects that group certain pods to offer access to them using a common name).
- **container runtime** - is the software responsible for actually running the containers in Kubernetes, such as containerd and CRI-O (previously also Docker).

- **container network and storage interface drivers (CNI and CSI drivers)** - drivers for network and storage solutions used in the Kubernetes cluster also run on the nodes.

Kubernetes extensibility

Kubernetes is highly extensible. There are multiple extension points. For example, there are the Container Networking Interface (CNI) and the Container Storage Interface (CSI), which allow for very flexible approach to networking and storage solutions. In the realm of security, the webhook-based authentication, authorization, and admission control allow for very fine-grained management of access policies. There are several other extension points, however, in the context of WP3, the most interesting is the extensibility of the API itself.

The API is organised around API groups, their versions and the (kinds of) objects they manage. Kubernetes allows to add custom resource definitions (CRDs; they are a kind of built-in object) which extend the API by adding new API group-version-kind triplets. Such custom resources can then be managed by standard Kubernetes tools, such as kubectl, like any other object of built-in kind. Coupled with custom controllers, which also can be deployed on Kubernetes itself, they offer a very powerful extension mechanism that allows for arbitrary actions to happen behind the scenes. This is now known as the “operator pattern” and several frameworks have grown to support this pattern. As an example, consider a custom resource that describes a database system – its supporting controller could manage instances of such a system: ensure the pods exist, that they are healthy, that they are properly scaled, etc., while the user is offered an entirely declarative and uniform way to configure it as if it was a native Kubernetes resource.

2.2 Modelling approaches supporting the generation of IaC

This section presents the modelling approaches that are relevant to DOML in order to study which DOML requirements could be satisfied by each approach.

2.2.1 CAMEL

In order to support the complete modelling of the user cloud applications, according to all the aspects needed, plus the models@runtime approach [23], the PaaSage project has created a super-DSL (Domain-Specific Language) called CAMEL (Cloud Application Modelling and Execution Language).

CAMEL includes multiple DSLs, each one focusing on a particular aspect of a cloud application modelling. CAMEL has been designed based on the EMF (Eclipse Modelling Framework) Ecore and OCL (Object Constraint Language). EMF Ecore enables the specification of UML-based meta-models, while OCL constraints accompany such meta-model specification with the coverage of additional domain semantics.

CAMEL has been derived from pre-existing languages, including CloudML [24] for the coverage of the deployment aspect, Saloon feature meta-model for the coverage of provider modelling, and CERIF (Common European Research Information Format) [18] for the organisation aspect coverage. Other sub-DSLs, like the SRL (Scalability Rule Language) [25], were developed in the PaaSage project to cover some missing aspects. All these sub-DSLs were integrated by moving them into the same technical space but also consolidating them to diminish their respective conceptual overlaps. Integration was also supported through the use of OCL rules focusing on cross-model validity (where cross-model means here “between models from different sub-DSLs but still within the same overall CAMEL model”).

Further development of the CAMEL language, including a node candidates' approach for resource modelling and utility function introduction, has been performed in the H2020 MELODIC project. Also, during that time, CAMEL has seen the first commercial usage. The development of

CAMEL is continuing within the H2020 MORPHEMIC project for both commercial and scientific purposes, focusing on the polymorphism aspects of cloud application modelling. This work will result in version 3 of the CAMEL language.

CAMEL is assorted, via the use of the Eclipse Environment, with supporting tools that ease the life of both developers and modelers. Developers are assisted through the creation of a programmatic API (domain model), via which, models conforming to CAMEL can be created and processed. Modelers are assisted via the offering of different editing tools. Such tools are either supplied by default by Eclipse, like the Eclipse tree-based editor, or available as add-ons. The latter tools include:

- (a) the CAMEL textual editor which supports the concrete syntax of CAMEL (realised via the Eclipse Xtext41 technology) and
- (b) the web-based editor which was developed based on the Eclipse RAP42 technology.

Figure 6 presents the high-level structure of CAMEL language and sub-DSLs.

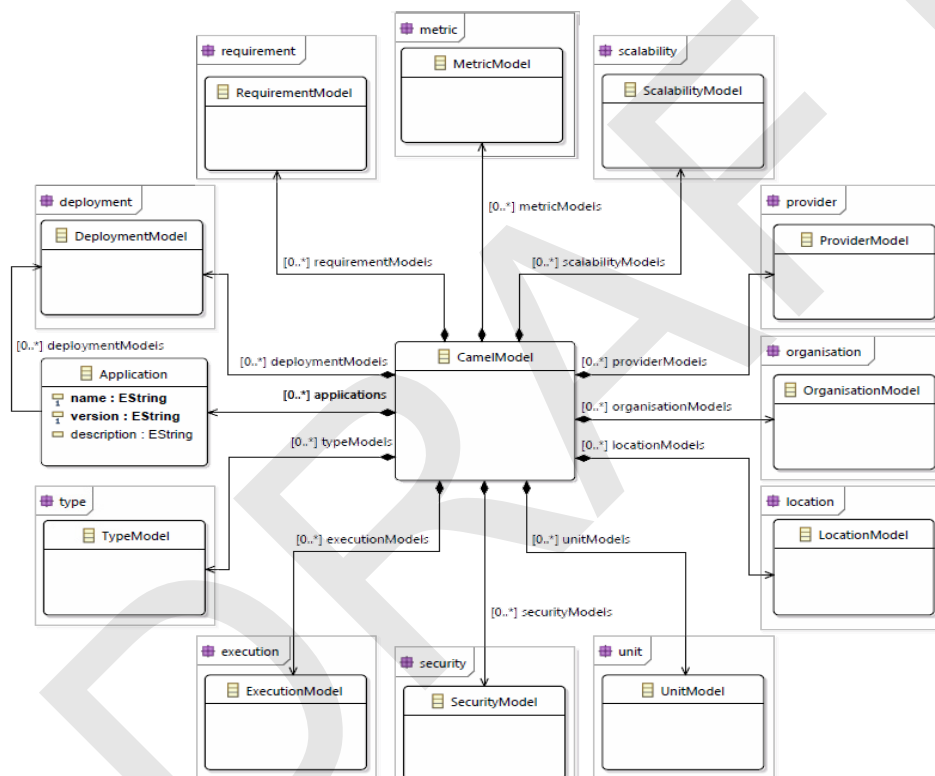


Figure 6. CAMEL main concepts

As an output, CAMEL is not producing IaC code to be executed, but the execution of the model is done through an orchestrating component. Currently, two orchestration frameworks can be used:

1. Cloudiator,
2. Activeeon ProActive Scheduler.

Both frameworks can execute the model described in CAMEL by provisioning infrastructure, deploying and managing application at runtime. Both frameworks also support application reconfiguration.

2.2.2 DICE Profile

DICE [26] is a DevOps platform for Big Data application development created by the DICE consortium through a collaborative effort sponsored by Horizon 2020. It includes both development and runtime tools, namely:

- DICE IDE: is an integrated development environment that enhancing coding, design and application prototyping based on the DICE Methodology and DICE Profile for UML.
- Quality analysis tools: these are a set of tools for quality analysis during the early-stage of application design through simulation, verification and optimization methods;
- Feedback and Iterative Enhancement tools: a monitoring platform based on the Big Data technologies and enhanced with tools for detecting quality anomalies that affect the application design and explain how data-intensive applications are using the resources to enhance the design;
- Continuous Delivery and Testing tools: these are tools and methods that support delivery on private and public clouds by a TOSCA-compliant deployment tool, application configuration, continuous integration, quality testing, and fault injection.

The central element of the DICE architecture is the Integrated Development Environment (IDE), where the developer specifies the data-intensive application using a model-driven engineering approach based on UML. The DICE UML profile provides the stereotypes as well as tagged values needed for the specification of data-intensive applications in UML.

Following a Model-Driven Architecture (MDA) approach, models are defined by the user in a top-down fashion. First the user provides platform-independent specifications of components and architecture (DICE Platform Independent Models, DPIM). Then he/she assigns specific technologies to implement such specifications (DICE Technology Specific Models, DTSM) and then maps the application components into a concrete TOSCA-compliant deployment specification (DICE Deployment Specific Models, DDSM) which can be realised as elaborate UML Diagrams or simple and deployment-specific DDSM images within the DICER stand-alone deployment modelling tool. Figure 7Error! Reference source not found. shows the main concepts of the top modelling layer (DPIM).

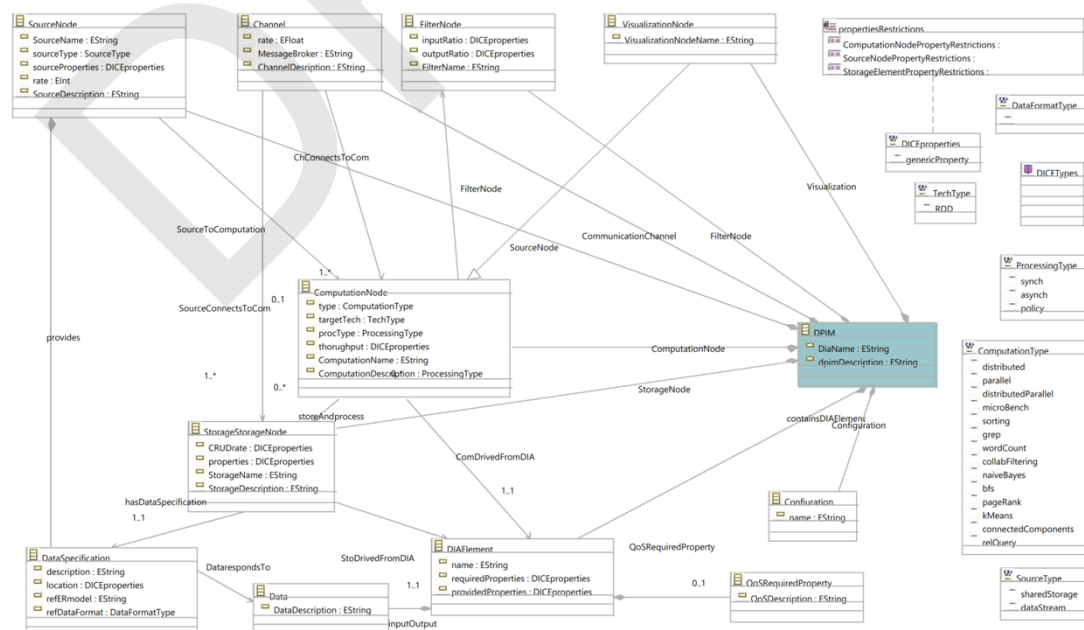


Figure 7. Data Intensive Applications concepts in DICE from [27].

Models at the various layers are related by DICE model-to-model transformations that are automatically executed within the IDE, to reduce the amount of manual work required from the user. The DICER tool enables the model-driven deployment of data-intensive applications (DIAs) taking into consideration the IaC paradigm. More specifically, DICER is also based on the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) standard and it is able to automatically generate IaC for DIAs in the form of TOSCA blueprints from stereotypes UML models.

DICE IDE gives the possibility to automatically translate DICE models into formal models for the assessment of quality properties (efficiency, costs, safety, correctness, etc.). Throughout the application design, each analysis requires to run dedicated tools which are supposed to be outside the IDE environment, in order to obtain prediction metrics. The simulation, optimization, and verification plugins take care of translating models across the IDE and these external tools. They also collect via REST APIs the outputs of these tools that are shown to the user inside the IDE. The interface of these plugins assumes the user to be unskilled in the usage of the formal models.

DICE monitoring platform (DMon) visualizes monitoring data in real-time from applications running on Big Data frameworks. Leveraging on Elastic.co's open-source technology stack, DMon is a fully distributed, highly available and horizontally scalable platform. All the core components of the platform have been wrapped in microservices accessible through HTTP RESTful APIs for easy control. DMon is able to monitor both the infrastructure (memory, CPU, disk, network etc.) and multiple Big Data frameworks, currently supported being Apache HDFS, YARN, Spark, Storm and MongoDB. The core components of the platform (Elasticsearch, Logstash, Kibana) and the node components running on the monitored cluster are easily controlled thanks to a Web-based user interface that backs up the DMon controller RESTful service. Visualization of collected data is fully customizable and can be structured in multiple dashboards tailored to specific roles, such as an administrator, quality assurance engineer, or software architect.

Through the optimization tool, the DICE architect assesses the performance and minimizes the deployment cost of data-intensive applications against user-defined properties, for instance, meeting deadlines under different levels of cluster congestion. The input is the DICE DDSM model, with targeted SLAs, and a description of the execution environment given as a list of possible providers, a list of VM types, etc. The optimization consists in finding the cheapest cluster configuration able to guarantee that application jobs are executed before a user-defined deadline. The architect can analyse the application behaviour under different conditions. For example, she can study the pros and cons of public clouds versus private clouds in terms of execution costs.

DICE Verification Tool allows designers to evaluate their design against user-defined properties as safety, reachability of undesired configurations of the system due to node failures or incorrect design of timing constraints. The verification process allows the DIA designer to perform verification tasks using a lightweight approach. The DICE Verification Tool approach is to perform a formal verification via interfaces that hide the complexity of the underlying models and engines. These interfaces allow the user to easily produce the formal model to be verified and the properties to be checked without the need of high technical expertise.

The Simulation Tool is a key component of the DICE Framework. The tool is able to simulate the behaviour of a DIA to assess its performance and reliability using a Petri net model. The DIA can be defined both at DPIM or DTSM level using a particular technology like Storm, Spark. The DIA needs to be modelled using the DICE profile.

2.2.3 SODALITE

SODALITE [28] is an EU H2020 project aiming at supplying tools for developers and infrastructure operators to abstract their applications and infrastructure requirements to facilitate the development and operation and execution of diverse applications over software-defined, high-performance cloud infrastructures. A model-based abstraction library as well as a design and programming model are developed for full-stack application and infrastructure descriptions. A deployment framework is established in order to support static optimization of the abstracted applications onto specific infrastructures, and automated run-time optimization and management of applications.

The SODALITE framework helps developers and infrastructure operators to simplify application and infrastructure description and suggest the best deployment scenarios, which enable DevOps teams to deploy their applications in a simpler and faster way.

The focus of SODALITE is to support the entire life cycle of the Infrastructure as Code (IaC). IaC helps to limit the need for DevOps teams to manually provision resources, configure them and deploy an application by offering the possibility to code such tasks into proper scripts that can be executed by specific orchestrators, thus introducing significant automation to the life cycle of the application.

The main innovations of SODALITE include:

1. Application Deployment Modeling and Infrastructure as Code Modeling - Building deployment patterns based on preexisting models: rule-based models, semantic models, and data-driven (machine learning and deep learning) models.
2. Ease of Deployment, Orchestration (xOpera), and Provisioning.
3. Verification and Bug Prediction - Support for a subset of smells, anti-patterns, and bugs in IaC scripts (TOSCA and Ansible).
4. Monitoring and Reconfiguration - Monitoring of metrics relevant for Cloud, HPC, and Edge computing environments and applications; basic event-driven deployment refactoring decision making; detecting and fixing performance anti-patterns.
5. Performance Optimization - Static optimization of applications for Cloud, HPC and Edge; Support for modelling performance.

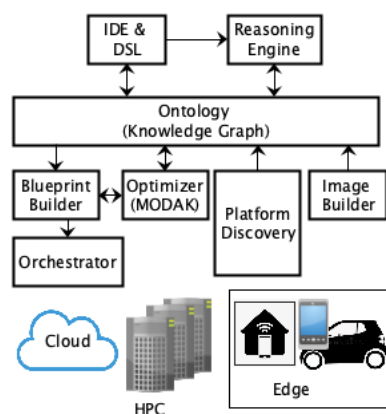


Figure 8. SODALITE high-level architecture of the design-time environment [29].

The high-level architecture of SODALITE is shown in *Figure 8* [29]. The Integrated Development Environment (IDE) facilitates the usage of a Domain Specific Language (DSL) to specify Abstract Application Deployment Models (AADMs). The Ontology is the basis for the definition and extension of DSL and enables the Reasoning Engine to offer advanced support to the user in terms of revealing possible errors in the AADM and possibilities for completion and extension. The Blueprint Builder is in charge of transforming the AADM into executable TOSCA and Ansible code. The Optimizer is able to select the execution container that appears to be most suitable for HPC applications. The Platform Discovery is in charge of automatically discovering and inserting in the Ontology information concerning new resources and the Image Builder of encapsulating application components into Docker containers. Finally, the Orchestrator is the one in charge of executing the deployment and redeployment steps based on the AADM definition. The platform includes also some runtime components, namely, a monitoring system and some components supporting autoscaling as well as refactoring and redeployment of AADMs.

The main results provided by SODALITE include:

- A pattern-based abstraction library that includes an application, infrastructure, and performance abstractions;
- A design and programming model (with IDE support) for both full-stack applications and infrastructures based on the abstraction library;
- An open-source deployment framework that supports the static optimization of abstracted applications onto specific infrastructures;
- Automated run-time optimization and management of applications.

SODALITE currently supports the application deployment on heterogeneous platform infrastructures such as HPC clusters managed by Torque and Slurm resource managers, OpenStack private/public cloud, public AWS EC2, and Kubernetes on Edge through Helm chart deployments. The DSL SODALITE offers is closely coupled to TOSCA and requires the development of Ansible code for the execution of specific configuration operations. A new DSL and editing support for this part is currently under development.

2.2.4 DECIDE modeling approach

In the H2020 DECIDE project [30], the aim was to provide tools for developers and operators to develop and operate multi-cloud native applications that can be dynamically self-adapted and re-deployed using the optimal combination of cloud services in each moment.

The DECIDE architecture [31] is shown in Figure 9, where the main DECIDE tools are positioned along the extended DevOps phases for multi-cloud applications defined in the project, i.e.

- 1- Architecting
- 2- Continuous development and testing
- 3- Pre-deployment
- 4- Continuous delivery
- 5- Continuous adaptation.

One element, that is depicted in a different colour to stress its importance is the Application Description (App Description in the Figure).

The Application description is neither a tool nor a component. It is a file where the actual status of the application is described.

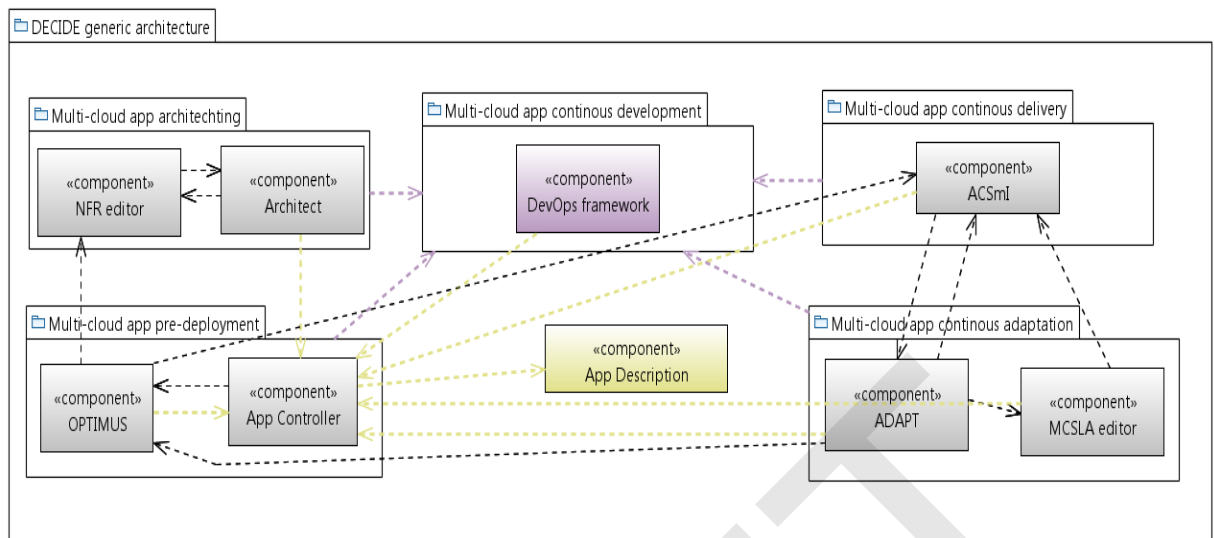


Figure 9. DECIDE integrated generic architecture [31].

The Application Description is the main mechanism to share information between tools and components in DECIDE framework. The Application Description is a structured JSON file, with a defined schema that serves to validate the file after each change. This file is used for the different tools in DECIDE to store/acquire relevant information with respect to the application, that is needed for the correct operation of each tool.

The Application Description represents a model of the application and of its deployment. It is stored in a git repository. The correct handling of this file is in charge of a java library (called Application Controller), that assists in managing the intelligence regarding the currently used deployment configuration and historical ones. It keeps records on whether a deployment configuration was successful and on whether any SLA violations had occurred during the application's operation time. With this information, the optimization module is able to suggest new and adequate deployment configurations, not reusing a previously unsuccessful deployment configuration.

The history file is located in a git repository adjacent to the Application Description. Both, the repository, and the Application Description are initially created during the Application development phase. The Application Description holds all necessary information for describing and classifying the application. It also holds the state of the application.

There can be potential problems when many tools work in parallel on one descriptor. Git repositories by nature are distributed. Each tool has to clone the repository and has to work on its local working copy. There exist challenges when tools work with distributed git repositories [32], as explained next.

First, they need to make sure that they work on the latest version, which means they probably should make a pull before starting a session. After they are done, the changes should be pushed back to the origin of the local clone, so other tools can see the result of the session. Both pull and push operations are potential sources for merge conflicts because the remote version could be changed between two pulls by other tools.

Usually, git should be able to merge the different versions without conflicts, but when there are conflicts, an appropriate strategy needs to be chosen for dealing with the situation. One major step to minimize possible merge conflicts is a well-designed structure of the descriptors, that

ensures that each tool needs to write exclusively in specific parts of the descriptors. The critical situation is where a tool is used in parallel by several users on the same project, that was not the case of the project. Each tool should start a session by pulling the last result from the remote repository, working on the local copy, and finally submitting the session result back. For this scenario, where only two branches need to be merged, the default git strategy *recursive* seems to be sufficient.

The Application Description can become a quite large file with many field and subfields. It contains items like microservices, resources, IP addresses, endpoints, Virtual Machines, SLAs, Containers, deployments patterns, etc. including every property needed to correctly define all these items. An excerpt of such a file can be seen in Figure 10.

```
{
  "id" : "57f69d92-f50a-48a3-936d-c637a8404cc5",
  "name" : "SockShopv2",
  "description" : "Another test",
  "version" : "0.2.2-alpha",
  "highTechnologicalRisk" : false,
  "preferredProvider" : "Amazon",
  "nfrs" : [ {
    "type" : "Location",
    "tags" : [ "Application" ],
    "abstractValue" : "Cross Border",
    "value" : {
      "region" : "Europe",
      "zone" : "Spain"
    }
  } ],
  "microservices" : [ {
    "id" : "eab14d56-387b-4f9e-9019-afb1959a3490",
    "name" : "front-end",
    "classification" : "Computing",
    "stateful" : false,
    "programmingLanguage" : "Java",
    "tags" : [ "Application" ],
    "publicIP" : false,
    "endpoints" : [ "" ],
    "detachableResources" : [ {
      "id" : "a35ab52d-a311-4eb8-8ff3-d36dc383f7d1",
      "name" : "none",
      "db" : false,
      "classification" : "storage"
    } ]
  }, {
    "id" : "8a995d87-d15d-456c-96eb-fc4aa5891bd2",
    "name" : "catalogue",
    "tags" : [ "Application" ],
    "publicIP" : false,
    "endpoints" : [ "" ],
    "detachableResources" : [ {
      "id" : "a35ab52d-a311-4eb8-8ff3-d36dc383f7d1",
      "name" : "none",
      "db" : false,
      "classification" : "storage"
    } ]
  } ]
}
```

Figure 10. Excerpt of an Application Description file.

2.2.5 RADON

RADON [33] is an EU H2020 project aiming to unlock the benefits of serverless Function as a Service (FaaS). The project introduces and advances DevOps framework that enables the creation and management of microservice-based applications. Serverless FaaS adoption is visible across many modern cloud environments such as AWS (AWS Lambda), Azure (Azure Functions) and GCP (Cloud Functions). Apart from responsiveness and highly parallel stateless execution, serverless computing also brings about challenges like code and data lock-in, early-stage adoption issues, portability and so on. RADON tackles these challenges by proposing a DevOps oriented framework that harmonizes the abstraction layers and optimizes decomposition and reuse. The framework is exposed through web-based multi-user development environment called Eclipse Che by using different RADON plugins that communicate with different services [34].

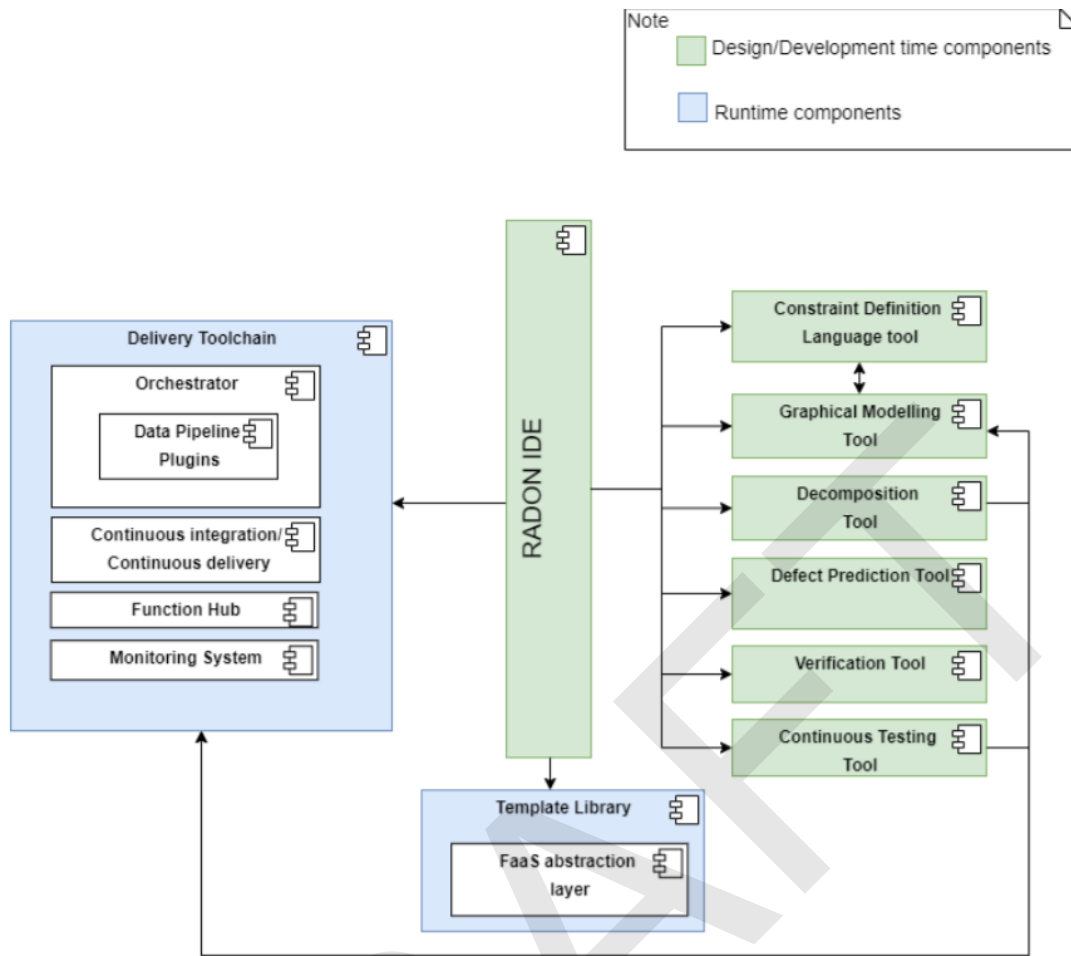


Figure 11. RADON architecture [34]

The main innovations and RADON tools are the following:

1. Graphical Modeling Tool is a web-based tool for graphical modelling of TOSCA applications.
2. Verification Tool can be used for the verification of IaC model and to find out whether it conforms to specified specification (defined using RADON Constraint Definition Language).
3. Decomposition Tool is used for architecture decomposition, deployment optimization, and accuracy enhancement.
4. Defect Prediction Tool can scan IaC and detect code smells.
5. Continuous Testing Tool enables continuous design, evolution, deployment, and execution of tests.
6. xOpera SaaS is a web-based orchestration tool for parsing and executing the RADON blueprints (i.e., TOSCA service templates packaged in a compressed archive called TOSCA CSAR).
7. Template Library introduces a shared repository of reusable TOSCA templates, blueprints, and modules required for the application deployment.
8. Monitoring System supports quality assurance by collecting the evidence from the runtime environment through a back-end service system.
9. Function Hub is a shared repository of versioned plug-and-play FaaS packages.

By using RADON tools, the users are able to prepare, validate and deploy their IaC templates. RADON blueprints mostly consist of TOSCA and Ansible, which are also present in PIACERE. The same applies for IaC scanning, where RADON Defect Prediction Tool can be used to check IaC correctness.

2.3 Generating IaC code from a model

The idea of generating code from an abstract model is at the heart of Model Driven Development (MDD), and dates back to the first years of this century [35]; but even before that, when in the '90s appeared the idea of Design Patterns, there was the idea to automatically generate the code to implement specific design patterns [36].

Since the beginning, the most used method for generating code from a model was the so-called Template-Based Code Generation (TBCG).

In the literature we can find several ways in which templates have been used for generating code from a model. In [37] the authors use Velocity [38] templates to generate Java code in the Fujaba [39] CASE tool. The proposed approach uses a further intermediate layer of tokens between the Abstract Syntax Graph (ASG) and the generated code. Tokens, each representing a code fragment to be generated, guide the template selection, and supply the parameters for the templates.

The tokens are first generated from the ASG, then a Sorting/Structuring phase decides which tokens are to be used and connects them in a graph, afterwards, an Optimization phase applies optimization patterns and restructures the token graph, and finally a Code Writing phase visits the token tree and generates the code for each node. One benefit of using templates for code generation is that details of the target language are expressed in the templates, thus making the generator quite independent from the target language itself.

It has been proposed [40] the projector templates for TBCG: templates that do not include control code and are inspired by the relational projection operator. A projective template is a hierarchy of textual blocks where each block contains text along with variables, called the block attributes, referencing corresponding model attributes to be substituted in their place. The proposed technology is included in the IDE tool offered by CodiScent [41].

A template is usually made up of a fixed part and a variable part. The fixed part contains some code in the target language that will be transferred verbatim into output, whereas the variable part may contain placeholders to be dynamically substituted by values and even control statements that can drive the generation process. Several Template Engine tools are available and each one has its own specificities.

In [42], the authors studied the Template-Based Code Generation (TBCG) tools used in the last 15 years. They identify the major characteristics of existing TBCG techniques, classify them, review the associated tools, and identify research trends in this field. The paper even provides a definition of TBCG: *“A synthesis technique is a TBCG if and only if it specifies a set of templates, assumes a design-time input, requires run-time inputs, and produces textual output”*. The authors found that TBCG is mainly used to generate source code and has been applied in a variety of domains.

To help in the selection of the right template engine to be used, the study from Luhunu [43] compares nine popular template-based code generation tools in terms of their expressiveness and performance. The paper distinguishes between model-based and code-based tools. Model-based tools (e.g., Acceleo, EGL) get their input from the model, whereas code-based (e.g. JET, Velocity) tools need to be fed by a specific programming language. The comparison concludes

that code-based tools are more performant, but model-based tools are more expressive. The table from the paper, comparing the main characteristics of the considered tools, is reported in Table 2, showing model-based tools in the first half and code-based tools in the second one.

Table 2. Classification of TBCG tools [43]

Tools	Dedicated language	Input type	Source domain	Execution mode	Typed block def.	Func.
Acceleo	MOFM2T extended OCL	Ecore model	Metamodel element	Interpreted	Yes	No
Xpand	Limited OCL	Ecore model	Metamodel element	Interpreted	Yes	No
EGL	EOL	Ecore model	Metamodel element	Interpreted	No	Yes
Xtend2	Abstraction of Java	Ecore model	Metamodel element	Interpreted or instant gen.	No	Yes
Velocity	Java-based scripting language	Java object	Any Java variable	Interpreted	No	No
JET	Java	Java object	Root of the data structure	Run Java output	No	No
T4	C#	C# object	Any C# variable	Instant gen. or Run C# output	No	Yes
String-Template	Limited scripting language	Java object	Any Java variable	Interpreted	No	No
XSLT	XSLT+XPath expressions	XML document	XML element	Interpreted	Yes	Yes

Practicing Model Driven Development, people soon discovered that it is not only useful to generate code from the model, but also operating the reverse transformation. For example, Bork et al. [44] proposed an approach to solve the reverse-engineering problem of obtaining the model back from the generated code. In the proposed approach, the templates used for generating the code, which may have been modified by the user for flexibility, are used as a "grammar" to parse the generated code. This approach has been implemented and tested in the Fujaba CASE tool.

The template-based technique used for code generation is the same as in [37] and uses a token graph as an intermediate language. Therefore, after pattern matching the code to find the right template and the value of its variables, the proposed approach recreates the tokens and then the original model from the token tree. Challenges in this approach are finding the right value for variables among multiple possible solutions, dealing with nested templates, and overcoming the inefficiencies of template pattern matching.

Actually, bidirectional model transformations can be considered a whole new research (sub)field. Stevens [45] provides some background for this field, analyses some issues and some properties that apply to transformations, such as models' consistency, when a transformation is bijective, what would happen to the source model when the target changes, if it should be automatic and under user control. In addition, transformation languages are explored, such as

QVT-R (for bidirectional transformations), MOF, XML-based options and graph transformations using Triple Graph Grammars (TGG).

TGG are another technique for model-based code generation, introduced by [46] and automated in the Fujaba tool suite [39]. Fujaba aimed from the start at supporting bidirectional transformations and declared it in its acronym: “From UML to Java and Back Again”.

The code generated in Model Driven Development has usually been programming language code, mostly in Java, but in other high-level languages as well. Only recently, some work appeared proposing the generation of code for an IaC language, like PIACERE aims to do. Sandobalin et al. [47] propose a Domain Specific Language (DSL) for modelling infrastructure and a tool named ARGON supporting this DSL and generating the Ansible scripts for provisioning an Amazon Web Services (AWS) platform.

The ARGON tool has then been extended [48] to support provisioning infrastructure on multiple clouds (Amazon and Azure) and to migrate from one cloud to another. The proposed Domain Specific Language provides abstractions for computing, storage, networking and elasticity, and includes a Provider-Independent Model (PIM) and a Provider-Specific Model (PSM), both specified using Ecore from the Eclipse Modeling Framework (EMF). To transform PIM into PSM, ARGON uses the ATL transformation language [49], which is used also to transform an Amazon PSM into an Azure PSM or vice versa. IaC code generation is done from PSM using Acceleo [50] templates.

The same authors recently conducted some experiments involving Computer Science students to compare the use of their Model-based tool ARGON with the direct use of Ansible to execute the same tasks [51]. The results of the study indicate that the model-based tool is both more effective and easier to use for the definition of the cloud infrastructure and accelerates the provisioning process.

2.4 Summary

The section discusses the state-of-the-art of IaC languages, modelling approaches and code generation techniques. For developing DOML, multiple different IaC languages might be used considering that some of them rely on each other, in order to complete the infrastructure provisioning, configuration and application deployment. For instance, TOSCA mainly describes the topology of an application, and at the same time it may need Ansible playbooks for the configuration. Moreover, the application may also use containers to host part of its components, which then require the corresponding docker script, etc. As for modelling languages, different approaches study the problem from different perspectives focusing on their specific objectives and supporting the generation of specific IaC languages. With respect to them, the DOML aims at being independent from specific IaC languages and at supporting multiple phases of the deployment and operation processes. However, the previous experience on modelling infrastructures and applications and on applying different techniques (e.g., xText, EMF, etc.) for developing the DSL is an important reference for the development of the DOML itself. For the IaC generation from DOML, the template-based code generation technique can be also adopted.

3 Requirements for the DOML

The requirements for the DOML are presented as part of Deliverable D2.1 [52]. They have been collected through multiple workshops with case study owners as well as plenary and peer to peer discussions with the technical partners. In fact, on the one side, the DOML should be able to describe the essential aspects of each case study. On the other side, it should provide configuration information to the runtime components of the PIACERE platform to allow them to work in the way expected by the specific application owner.

In this section we provide a summary of the DOML-specific requirements grouped in two tables, one listing requirements on the general characteristics of the DOML (Table 3) and another concerning the elements of applications and infrastructures the DOML should represent (Table 4). **Error! Reference source not found..** 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 followed by the more specific ones. Of course, for traceability reasons, the requirement identifiers defined within the analysis carried out as part of Deliverable 2.1 have been kept.

Table 3. Requirements on the general characteristics of the DOML [52].

Req ID	Description	How DOML is addressing this requirement
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.
REQ62	DOML must support different views.	As it will be discussed later on in this document, DOML allows models to be defined on a per-layer basis. Layers represent different viewpoints on the system. More specifically, the viewpoints that are taken into consideration enable: <ol style="list-style-type: none"> 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.
REQ70	The DOML should allow users to state correctness properties in a suitable sub-language (possibly Formal Logic).	This requirement is addressed in the DOML in two different ways: <ol style="list-style-type: none"> 1) the main correctness relationships among elements in the specification are offered directly as part of the language and can be used, for instance, to express typical properties such as: a component is running within a certain execution environment or an execution environment belongs to a certain network. between elements.

Req ID	Description	How DOML is addressing this requirement
		2) the language offers the possibility to express generic constraints on the elements that are used in a certain DOML model. This last part will be further developed in the next iterations of the language definition.
REQ76	DOML should allow the user to model each of the four considered DevOps activities (Provisioning, Configuration, Deployment, Orchestration).	The DOML language offers the possibility to define models that can be used in all aforementioned activities. At the moment, it does not cover the possibility to define workflows where the mentioned activities can be executed in different orders in reference to different elements of a DOML model. This last aspect will be reconsidered in the next iteration of the language definition on the basis of the specific examples that will be identified.
REQ57	It is desirable to enable both forward and backward translations from DOML to IaC and vice versa	The main objective of DOML is to enable the forward translation into IaC. The ongoing work within WP3 is focused on this specific activity. However, 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, and the main forward translations will be working properly.

Table 4. Requirements on the specific elements to be modelled in the 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).
REQ25	DOML should support the modelling of security rules (eg. by type tcp/udp., and ingress/egress port definition)	This requirement is addressed by proper elements offered in the language. An evaluation of the offered constructs is still pending and will be performed in the following months.
REQ26	DOML should support the modelling of security groups (containers for security rules)	This requirement is addressed by a specific construct in the language.
REQ27	DOML should support the modelling, provisioning,	The DOML addresses this requirement by offering constructs to define a container, a container image and a container file.

Req ID	Description	How DOML is addressing this requirement
	configuration, and usage of container engine execution technologies (eg. docker-host)	
REQ28	DOML should support the modelling of containerized application deployment (e.g. pull/run/restart/stop docker containers)	As stated for REQ27, the DOML offers the possibility to model containers and its constituents. As stated for REQ76, for the moment it does not support the modelling of workflows to which the pull/run/restart/stop activities belong to.
REQ29	DOML should support the modelling of VM provisioning for different platforms such as (OpenStack, AWS) for canary and production environments	The DOML offers the possibility to define different types of VMs.
REQ30	DOML should enable support for policy definition constraints for QoS/NFR requirements	DOML supports the definition on QoS/NFR requirements (see REQ61). In the next months 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 modeling abstractions to define the outcomes of the IoP	The outcome of IoP is a mapping between a partial DOML model specified by the user and some concrete instances of infrastructural elements. The DOML offers the possibility to represent this mapping through specific relationships between elements. A completely mapped model will result into the generation of a fully executable IaC.
REQ59	The DOML should allow users to define rules and constraints for redeployment, reconfiguration and other mitigation actions	The DOML at the moment addresses this requirement by supporting the definition of requirements and constraints that should be taken into account while performing mitigation actions. These concern, for instance: <ul style="list-style-type: none"> 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.
REQ60	DOML should support the modelling of security metrics both at the level	For the moment this aspect is not explicitly addressed in the DOML. However, the sub-language used for defining generic non-functional requirements could be suitable to address the modelling of security metrics as

Req ID	Description	How DOML is addressing this requirement
	of infrastructure and application	well. Experiments will be conducted in the following months and the language will be extended if needed.
REQ61	DOML must support the modeling of NFRs and of SLOs	The DOML supports the definition of non-functional requirements predicating on response time, availability, or other characteristics of application components.
REQ36	DOML to enable writing infrastructure tests.	For the moment this aspect is not addressed in the DOML. However, the definition of DOML model could guide either a human being or an intelligent software in the definition of an infrastructure testing plan. Such plans 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 in the second project year.

4 DOML modelling principles

The development of the DOML is taking into account all requirements defined in the previous section as well as two guiding principles that have motivated the development of the PIACERE project since its inception. In this section we present these principles.

4.1 A single model for multiple IaC code fragments

As it should be clear from the analysis of the state of the art presented in section 2, modelling approaches aiming at supporting users in the definition of IaC do exist. They are based on the idea of offering the end user some high-level notation that maps into a specific IaC language which is focusing mainly on some specific tasks. For instance, all approaches that map into TOSCA are focusing on defining the topology of a software system with the purpose of orchestrating the provisioning of the VMs needed for this system as well as the black-box execution of other scripts (e.g., Ansible playbooks) that are focusing on the more complex operations which, for instance, deploy the needed software layers on top of the provisioned VMs. These scripts are meant to be provided by the end users while the modelling approach supports exclusively their connection to specific portions of the main model.

In the design of DOML, instead, we would like to keep the usage of external scripts as low as possible allowing the users to create models that can result in IaC code written in different languages and dedicated to executing different operations.

For instance, let's assume that we create a DOML model corresponding to the UML diagram shown in Figure 12 (for simplicity, we are adopting here a well-known notation as UML to be able to formulate examples in an intuitive way; the syntax of the DOML, however, will not be based on UML to avoid to cope with its complexity). The diagram shows a component called A that requires the installation of NodeJS for its execution. In turn, NodeJS is running on a Docker container on a certain VM.

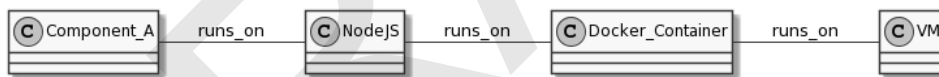


Figure 12. Relationships between a component and the execution environment it runs on

From this example, we can infer that the following steps should be performed when one wants to deploy and run the system:

1. A container image will have to be created and will have to incorporate NodeJS and component A.
2. A VM with the required characteristics will have to be provisioned and associated to a proper public IP address; this step can be executed in parallel with respect to the previous one.
3. A Docker engine will have to be deployed in this VM, if not already present in the corresponding image.
4. The created container image will have to be run on the VM, in the context of the Docker engine.

If the container is properly configured, at this point the web server and our component A will start their execution.

The above steps can be accomplished, provided that we generate, either manually or automatically the following artifacts:

- A Dockerfile in charge of managing the creation of the container image (step 1)

- A Terraform or TOSCA blueprint in charge of orchestrating steps 2 to 4, interacting with the VM provider and executing all needed scripts.
- Some Ansible playbooks or similar scripts that execute steps 3 and 4.

Besides the complexity of each of the individual files to be created, an important issue we note is that these files are all written using different languages featuring a different programming model. With DOML we would like to understand the extent to which the scripts needed to accomplish the majority of the above steps can be derived from a high-level model that includes the elements identified in Figure 12, thus limiting as much as possible the need for the end users to explicitly work with the target languages.

4.2 Separation of concerns and multiple modelling layers

Another important objective we want to tackle is to support users in separating the modelling of application-level components from the one of their execution environments (e.g., containers, VMs, ...). The rationale for this choice is that different users could be focusing on these two aspects. In fact, typically, the application designer will focus on the definition of the application structure in terms of components and their connections (see Figure 13), while an Ops expert will be in charge of allocating components within proper computational elements. Such allocation will have to allow the fulfilment of the specified non-functional requirements. Finally, the DevOps expert will create proper configurations of some computational elements (see Figure 14).



Figure 13. Modelling the application structure

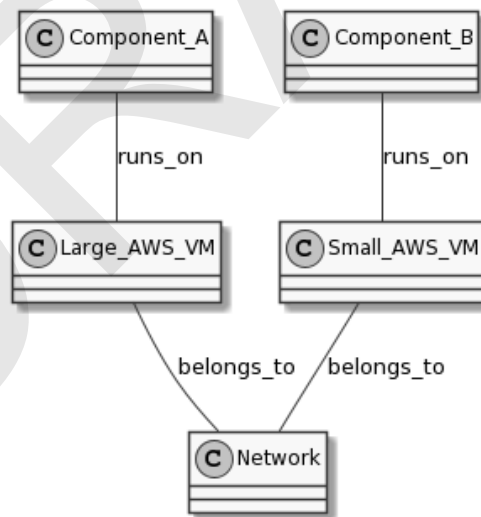


Figure 14. Modelling the infrastructure and the mapping with components

Furthermore, given the availability nowadays of multiple providers/technologies offering IaaS (Infrastructure as a Service) and, in some cases, compatible PaaS (Platform as a Service) solutions, we want to offer the possibility to provide an abstract definition of the infrastructure to be used for a certain application and, then, to define different *concretizations* of this same infrastructure, so to support deployment and execution of applications into multiple contexts.

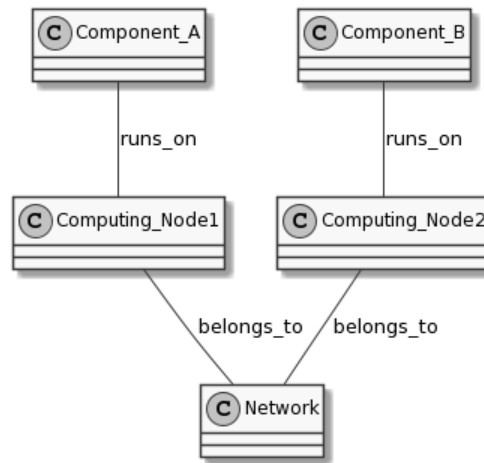


Figure 15. Modelling an abstract infrastructure and the mapping with components

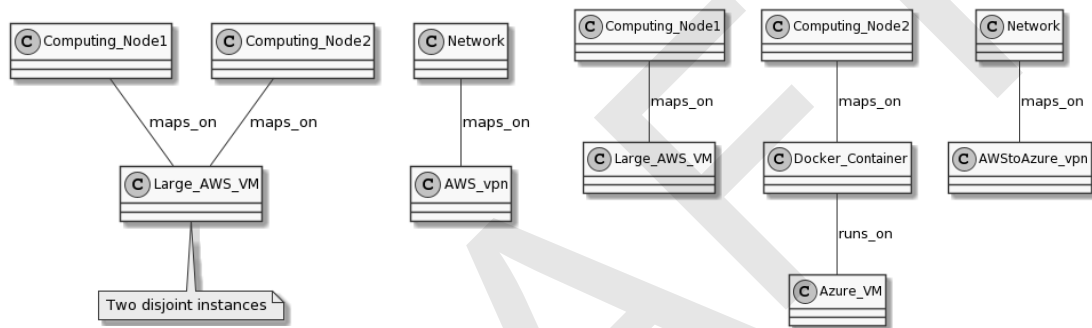


Figure 16. Modelling different concretizations of an abstract infrastructure

This requirement has emerged from the discussion with one of our use case owners as they want to allow for different deployments of the same case: an in-house containerized installation of the product to be used for pre-release testing and a cloud-based non-containerized installation to be used as main operational environment.

To accommodate all above aspects, the DOML allows the user to define a mapping between application components and abstract infrastructural elements (see Figure 15) and then derive from this, either manually or by exploiting an automated tool (IOP or VT in the PIACERE case) some possible concretizations (see Figure 16).

5 Modelling abstractions relevant for IaC definition

In this section we provide an overview of the modelling abstractions we have identified. The section starts with a presentation of the approach we have adopted to identify these abstraction (see Section 5.1). Then the four packages that aggregate the modelling abstractions in coherent groups are presented (Sections from 5.2 to 5.6).

5.1 Adopted approach

The identification of the elements to be included in the DOML has been conducted analysing the literature (see Section 2) and gathering information from all PIACERE partners following an iterative process. For what concerns this second aspect, the analysis was conducted as follows:

Acquisition of information by partners about the concepts to be included in the DOML: we asked all partners to include in a shared document all elements that they thought had to be represented in DOML. For each element we requested the following pieces of information: description of the element, rationale for including the element in the DOML, relationships with other concepts in the DOML, relevant DOML layer (to be selected among the following: application layer, abstract infrastructure layer, concrete infrastructure layer), relevant software lifecycle phase (to be selected among the following phases derived from the analysis of the state of the art: verification, IaC generation, testing, provisioning, infrastructure configuration, application deployment, application configuration, optimization, monitoring, self-healing; it was also possible to select the all phases option), and example of usage.

Analysis of acquired elements, elimination of duplicates and discussion about unclear concepts: In this phase we highlighted the concepts that seemed to be replicated or related among each other and discussed them with all partners. At the same time, we realized that the analysis of individual concepts was difficult for our partners as they needed to visualize the context in which a certain concept was introduced and how it was related with the other concepts. For this reason, we moved on to the next step.

Definition of UML diagrams to characterize the concepts as entities, entity attributes and relationships between entities: starting from the identified concepts we built a UML diagram including all identified entities and highlighting the relationships between them. At the same time, we developed an example in UML modeling the deployment of a WordPress installation. This example has been used in [53] to compare to different IaC approaches, one adopting Terraform and the other the TOSCA dialect adopted by Cloudify. Thanks to this research work, we could get access to the IaC code made available by the authors and we used it as a reference target for our modeling example.

Discussion and reiteration on the UML diagram: The UML diagrams have allowed us to gather additional feedback by the partners. Additional details have been added and clarified.

Organization of the diagram in different packages pertaining the following aspects: to cope with the relatively large number of concepts we have organized the concepts in the following packages:

- *Application layer:* this includes the concepts associated to the definition of an application, that is, the concepts of component, interface, connector between component, service, and specific subcategories of components and services, e.g., a DBMS or a GPS location service.
- *Abstract infrastructure layer:* this includes the concepts associated to the definition of the infrastructure, without referring to a specific provider or vendor, for instance, the concept of computing node.

- *Concrete infrastructure layer*: this includes concepts associated to the definition of specific infrastructure elements, e.g., a Docker container or an Amazon VM.
- *Optimization*: this includes concepts that are needed by the IoP component to find an optimal realization for a DOML. The most prominent element is the concept of optimization objective which can consists in the minimization or maximization of certain properties.
- *Common elements*: this includes those elements that can be used in more than one of the above layers. Examples are the concepts of requirement and properties.

In the following we introduce the relevant diagrams defined for each of the above packages while their detailed definition is available in the Annex [1]. The reader should note that, even if these diagrams went through three revisions so far, they are not to be considered as final at this stage and will be enhanced and expanded during the PIACERE project based on the inputs we will receive from all partners.

5.2 Common elements

As defined above, this layer includes the common elements of the DOML metamodel used in other layers (see Figure 17).

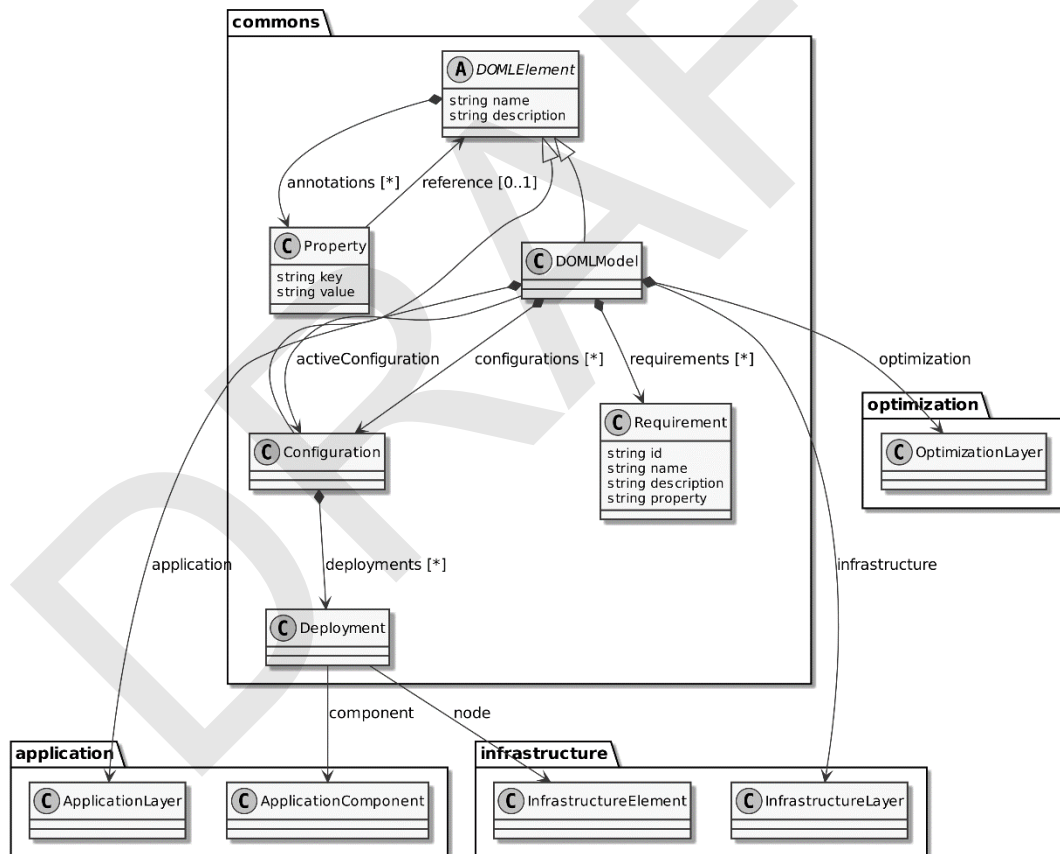


Figure 17. Structure of the Common Elements layer

Specifically, this layer contains the base *DOMLElement* which has different *Properties* and two concrete extensions: the *Configuration* and the *DOMLModel*. A *Configuration* can be composed of different *Deployments* of components (*ApplicationComponent* in application layer) and nodes (*InfrastructureElement* in abstract infrastructure layer). Then, a *DOMLModel* can be composed of different applications (specified in the *ApplicationLayer*), configurations, infrastructures (specified in the *InfrastructureLayer*), requirements and optimization (specified in the

OptimizationLayer). Details about the *ApplicationLayer*, *InfrastructureLayer*, and *OptimizationLayer* are described in the following sections.

5.3 Application layer

In the above section, the connection between the Common elements layer and the application layer has been identified. Hereafter, we discuss the detail structure inside the application layer.

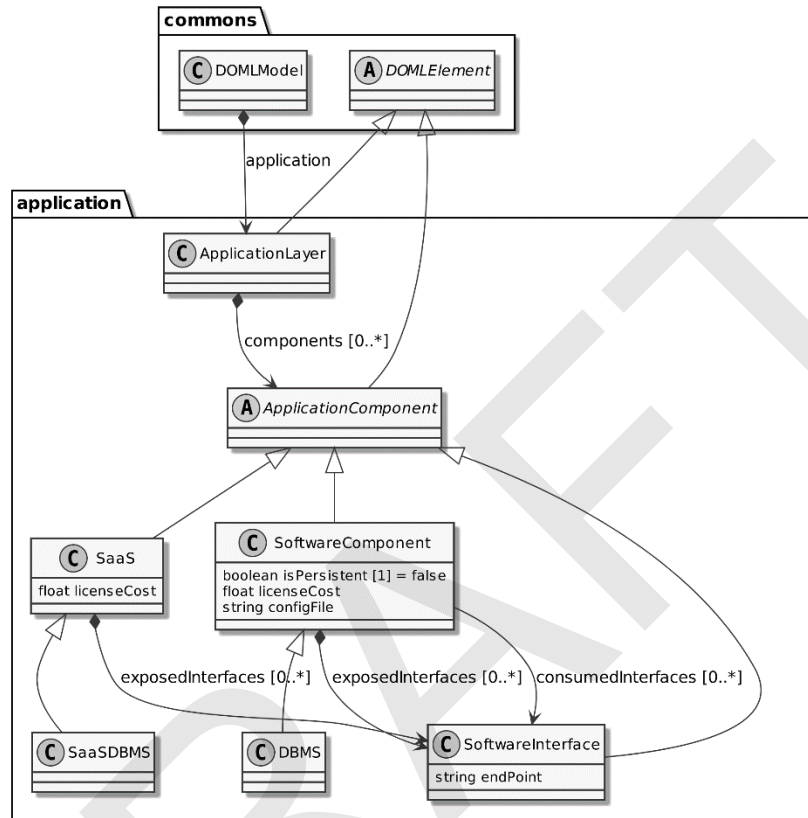


Figure 18. Structure of the Application layer

In the application package (see Figure 18), an *ApplicationLayer* can be composed of many *ApplicationComponents*, which can be a *SaaS* (Software as a Service) component, a *SoftwareComponent*, or a *SoftwareInterface*. Among the different application components, a *SaaS* can have the *SaaSDBMS* extension and the *SoftwareInterface*. A *SoftwareComponent* can have the *DBMS* and expose its own software interface or consume the interface provided by other software components.

5.4 Abstract infrastructure layer

In the Common elements layer, we have briefly introduced the relationship with the abstract infrastructure layer. Here we discuss many different kinds of *DOMLElements* inside the abstract infrastructure layer and their connections.

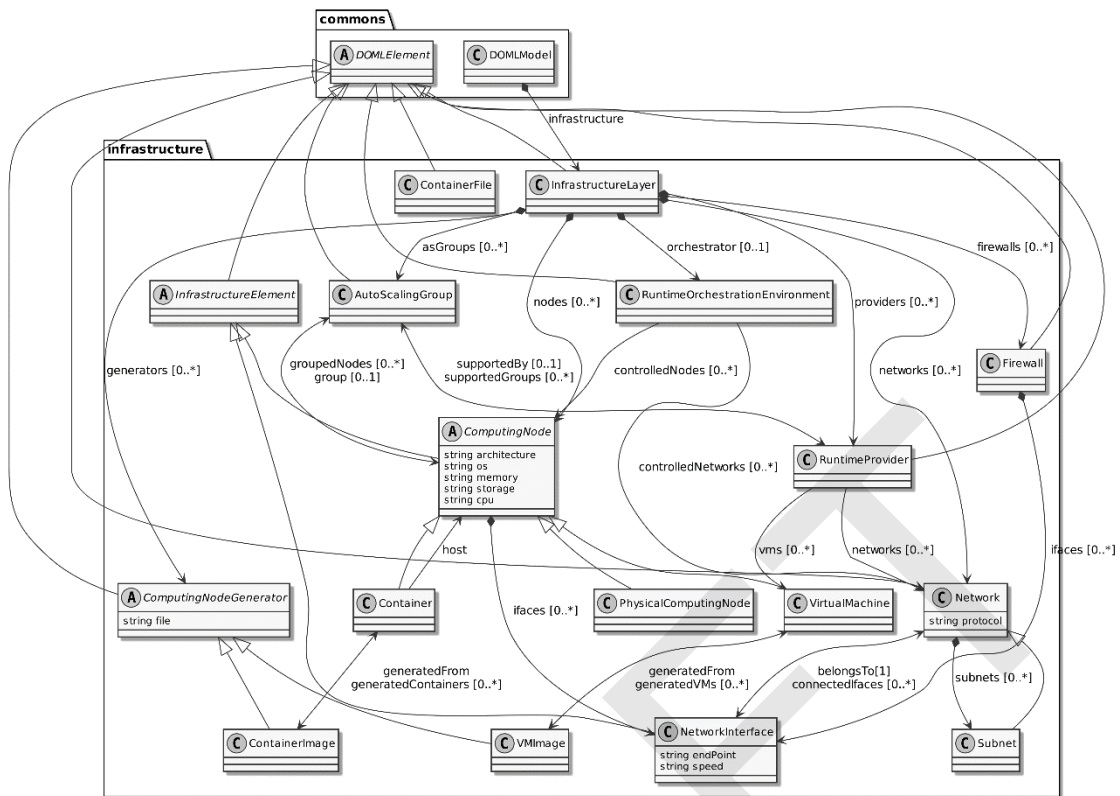


Figure 19. Structure of the Abstract Infrastructure layer

More specifically, as a type of *DOMLElement*, the *InfrastructureLayer* (see Figure 19) is composed of the *ComputingNode*, *ComputingNodeGenerator*, *Network*, *Firewall*, *RuntimeProvider*, *AutoScalingGroup*, and *RuntimeOrchestrationEnvironment*. The *InfrastructureElement* is another type of *DOMLElement*, which can be a *ComputingNode* or a *NetworkInterface*. A *ComputingNode* can have multiple *NetworkInterfaces*, and the concrete type can be a *Container*, a *PhysicalComputingNode* or a *VirtualMachine*, where the *Container* is generated from the *ContainerImage* and the *VirtualMachine* is generated from the *VMImage*. Both *ContainerImage* and *VMImage* play the roles of *ComputingNodeGenerator*. A *Network* can have many *Subnets* and *NetworkInterfaces* which connects different types of *ComputingNodes*. The *Firewall* is associated with the *NetworkInterface*. Another import type of *DOMLElement* is the *RuntimeProvider* which provides the *VirtualMachine*, *Network* and supports different *AutoScalingGroups* for the *ComputingNode*. Finally, the *InfrastructureLayer* can also have a *RuntimeOrchestrationEnvironment* controlling the computing nodes and the networks.

5.5 Concrete infrastructure layer

This layer provides the concretization for the abstract infrastructure layer with the specific infrastructure elements. This layer will be continuously extended with the insertion of new concrete elements by exploiting the DOML extension mechanism (DOML-E).

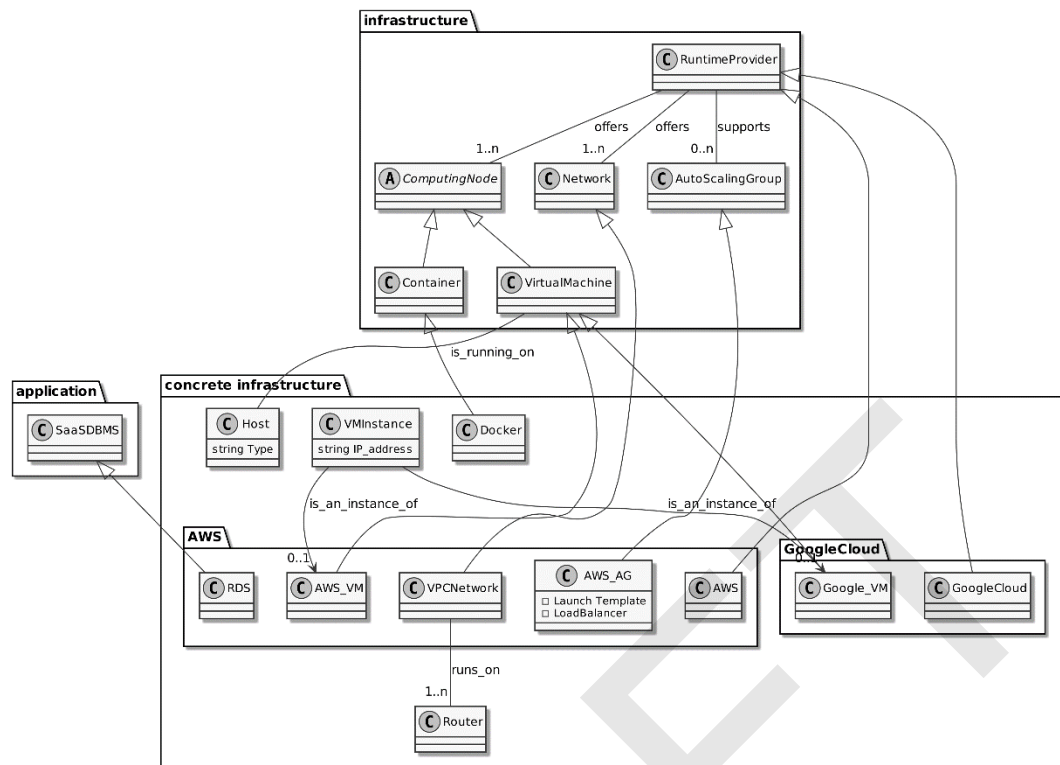


Figure 20. Structure of the Concrete Infrastructure layer

Specifically, the currently available concrete infrastructure is shown in Figure 20 and mainly includes the *RuntimeProviders* (e.g., AWS and Google Cloud) and the *Docker Container*. Note that here the concrete infrastructure layer only includes two providers but will support more in the next stage of DOML. The mapping between the abstract and concrete infrastructures are shown in the above figure. In the concrete layer, *AWS_VM* and *Google_VM* are the two provider-specific *VirtualMachines* with their own *VMInstances*. The *VirtualMachine* also runs on the *Host* like a specific physical machine. Finally, other elements in AWS package include: *RDS* providing the service *SaaSDBMS* in application layer, a *VPCNetwork* running on a *Router* and an autoscaling group *AWS_AG* for controlling the scalability of nodes.

5.6 Optimization

The optimization layer mainly provides the information for the IOP component which aims at finding a best infrastructure resource configurations and deployments.

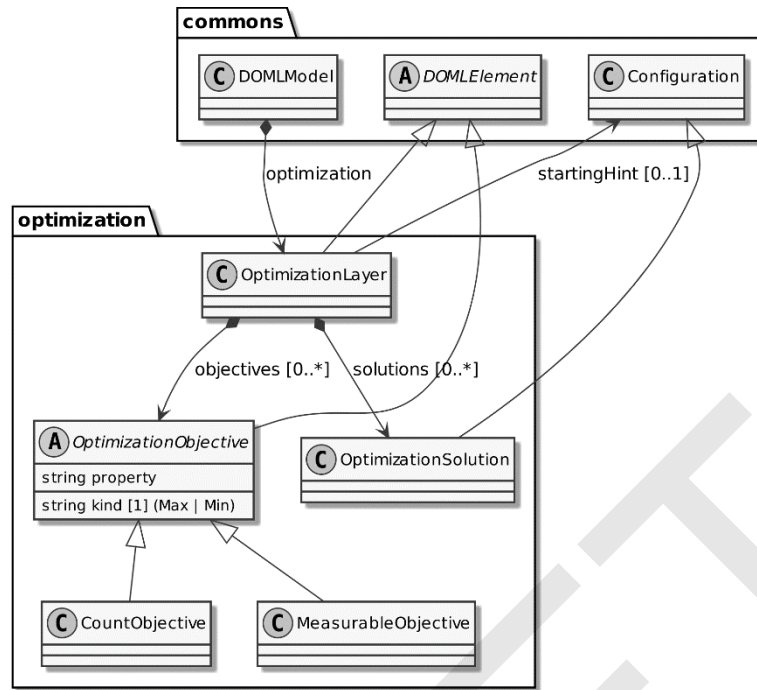


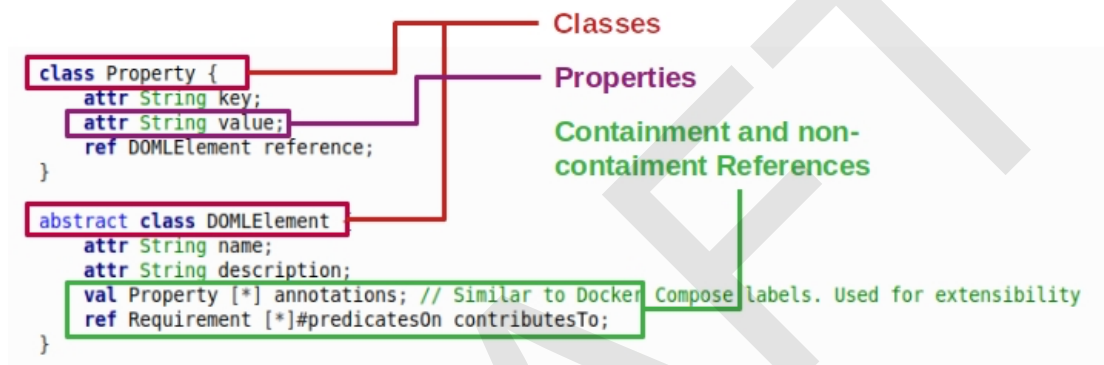
Figure 21. Structure of the Optimization layer.

Specifically, as a concrete type of the *DOMLElement* and a part of the *DOMLModel*, the *OptimizationLayer* can define multiple *OptimizationObjectives* which could be *CountObjective* or *MeasurableObjective* (see Figure 21). After the optimization process of PIACERE, the *OptimizationLayer* could provide multiple feasible *OptimizationSolutions* that could be used for the configuration of the infrastructure resources.

6 The DOML v0.1 language

For the implementation of the DOML language the modelling technologies present in Eclipse has been used, as Eclipse is the selected framework for the implementation of the PIACERE IDE. The language has two different parts: the actual DOML metamodel, described using Ecore (the Eclipse implementation of the Meta-Object Facility language MOF) and the text syntax used to create models. The current DOML text syntax is exemplified in the examples shown in Section 7 and is defined in the Annex.

For the implementation of the DOML, the Emfatic Ecore editor [54] has been chosen, as it provides a textual doorway to EMF which is more appropriate for teamwork and version control. Metamodels in Ecore, are defined in terms of metaclasses, with a set of attributes and references to other metaclasses. In the following excerpt of the DOML definition, all of the latter features are shown:



As depicted in the figure, metaclasses are defined similarly to Java/C++ classes, and their properties and relations are defined by using the **attr**, **val** and **ref** keywords. The DOML metamodel implementation, which is available in the git repository of the PIACERE project, is fully described in the Annex, where each of the metaclasses, their usages, alongside with their properties and relationships are described. The Annex document is to be treated as the specification of the DOML language, and it is therefore provided as a separate document for usability purposes.

In order to build DOML models, different editors will be provided as part of the PIACERE framework. These editors will be linked to the EMF tools in the IDE and will provide a gateway to the DOML. The DOML models shown in this document are using a tentative syntax, but this syntax must not be taken as the final one. Instead, the goal of the DOML syntax used in this document is to better understand the information inside the DOML.

7 First DOML examples

In this section, we illustrate how two simple, but realistic applications and their supporting infrastructure can be specified through DOML.

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 deployment infrastructure we describe in DOML consists of:

- Two load-balanced servers running WordPress.
- A SQL database.
- The network infrastructure required to link them.

We present two versions of this deployment, differing in the implementation chosen for the SQL database. In the first one, described in Section 7.1.1, an instance of the Postgres DBMS runs on a virtual machine automatically configured by the ICG through an existing image. In the second one, described in Section 7.1.2, the database is provided as SaaS.

7.1.1 DOML model with a VM database

In Figure 22, we show a diagram of the cloud infrastructure deployed by this example.

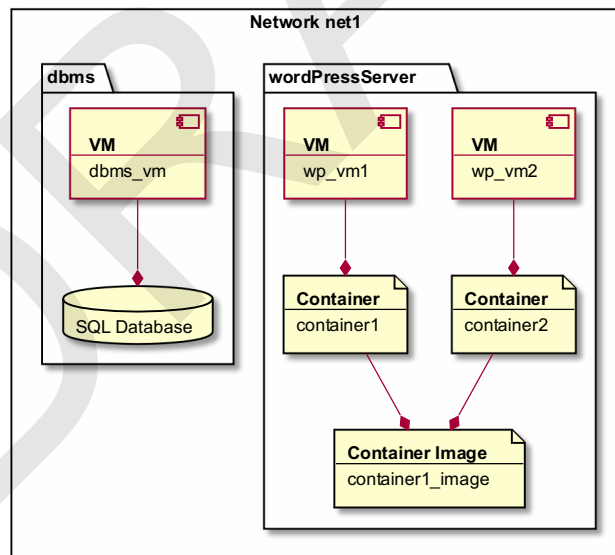


Figure 22. Diagram of the WordPress example with a VM database

We report the code of the example below, divided into several code snippets, which we explain separately. In practice, all such code snippets are merged in a single DOML file.

```
doml wordpress
```

In the first line of the file, we declare the name of the DOML model.

7.1.1.1 Application Layer

Next, we define the application layer.

```
application app {
  dbms database {
    provides {
      SQL_interface
    }
    properties {
      identifier      = "education";
      engine          = "postgres";
      engine_version  = "13.1";
      username        = "danilo";
      password        = "passw0rd";
      name            = "wp_db";
    }
  }

  software_component wordpressServer {
    consumes {
      SQL_interface
    }
  }
}
```

The application layer consists of two components: the database and the WordPress server. Here we declare the relationships between them: the database provides a SQL interface, which is consumed by the WordPress server. Additionally, we define here the properties of the DBMS that are agnostic with respect to its implementation in the infrastructure layers: namely, we declare the DBMS software and version, and the access credentials. Note that, since this is a preliminary version of DOML, user input and external references are not supported yet. Thus, the database credentials are reported verbatim, although this is a bad practice that may cause security issues. In the next DOML versions, we plan to allow for references to user-supplied values, so that credentials do not have to be hard-coded.

7.1.1.2 Abstract Infrastructure Layer

The abstract infrastructure layer is shown below:

```
infrastructure infra {
  container container1 { }
  container container2 { }
  cont_image container1_image {
    generates container1, container2
  }
  vm wp_vm1 {
    ifaces {
      iface i1 {
        address "10.0.1.3"
      }
    }
  }
  vm wp_vm2 {
    ifaces {
      iface i1 {
        address "10.0.1.1"
      }
    }
  }
  vm dbms_vm {
    ifaces {
      iface i1 {
        address "10.0.1.2"
      }
    }
  }
}
```

```

    }
    net net1 {
        address "10.0.1.0/24"
        protocol "tcp/ip"
    }
    autoscale_group g1 { wp_vm1, wp_vm2 }
}

```

Here we define the two VMs that will run WordPress (wp_vm1 and wp_vm2) and the one running the DBMS (dbms_vm) and assign them custom IP addresses. The two WordPress VMs are included into an “autoscale group”, which means that client requests will be automatically balanced between them. We also define the internal network that allows the WordPress servers to communicate with the database. Moreover, the WordPress instances will be deployed through containers sharing the same container image, which will be equipped with all the software needed to run WordPress.

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

```

deployment config1 {
    container1 -> wp_vm1,
    container2 -> wp_vm2,
    wordpressServer -> container1,
    wordpressServer -> container2,
    database -> dbms_vm
}
active config1

```

Containers are also linked to the VMs they will run on. The last line states that the deployment configuration above is currently active. In principle, multiple deployment configurations could be defined and switched.

7.1.1.3 Concrete Infrastructure Layer

The abstract infrastructure can be concretized in two ways: con_infra1 uses AWS as a cloud provider, while con_infra2 uses GCP. We show them below:

```

concretizations {
    concrete_infrastructure con_infra1 {
        provider aws {
            vm_image ubuntu_ami {
                properties {
                    most_recent = true;
                    name_regex = "ubuntu*";
                    owners = ["099720109477"]; // Canonical
                }
            }
        }
        vm concrete_vm1 {
            properties {
                ami = "ubuntu_ami";
                instance_type = "t2.micro";
            }
            maps wp_vm1
        }
        vm concrete_vm2 {
            properties {
                ami = "ubuntu_ami";
                instance_type = "t2.large";
            }
            maps wp_vm2
        }
    }
}

```

```

vm concrete_dbms_vm {
  properties {
    ami = "ubuntu_ami";
    instance_type = "db.t3.micro";
    allocated_storage = 1;
    vpc_security_group_ids = ?;
    db_subnet_group_name = "concrete_net";
    publicly_accessible = true;
    skip_final_snapshot = true;
  }
  maps dbms_vm
}

net vpc_network {
  properties {
    cidr = "10.0.0.0/16";
    name = "piacere_vpc";
  }
}

subnet concrete_net {
  properties {
    cidr_block => net1;
    vpc_id = "vpc_network";
    name = "piacere_subnet";
  }
  maps net1
}

}

provider docker {
  cont_image concrete_wp_image {
    properties {
      name = wordpress;
      image = wordpress:5.8.0;
      ports = "8080:80";
      env {
        WORDPRESS_DB_HOST => infra.dbms_vm;
        WORDPRESS_DB_USER => app.database.username;
        WORDPRESS_DB_PASSWORD => app.database.password;
        WORDPRESS_DB_NAME => app.database.name;
      }
    }
    maps container1_image
  }
}

}

concrete_infrastructure con_infra2 {
  provider gcp {
    vm concrete_vm1 {
      properties {
        name = "name";
        machine_type = "machine_type";
        zone = "zone";

        boot_disk {
          initialize_params {
            image = "debian-cloud/debian-9";
          }
        }

        scratch_disk {
          interface = "SCSI";
        }

        network_interface {
          network = "default";

```

```
        access_config { }
    }
}
maps wp_vm1
}

vm concrete_vm2 {
  properties {
    name          = "name";
    machine_type  = "machine_type";
    zone          = "zone";

    boot_disk {
      initialize_params {
        image = "debian-cloud/debian-9";
      }
    }

    scratch_disk {
      interface = "SCSI";
    }

    network_interface {
      network = "default";
      access_config { }
    }
  }
  maps wp_vm2
}

vm concrete_dbms_vm {
  properties {
    name          = "name";
    machine_type  = "machine_type";
    zone          = "zone";

    boot_disk {
      initialize_params {
        image = "debian-cloud/debian-9";
      }
    }

    scratch_disk {
      interface = "SCSI";
    }

    network_interface {
      network = "default";
      access_config { }
    }

    settings {
      tier = "tier";
    }
    deletion_protection = "deletion_protection_value";
  }
  maps dbms_vm
}

net vpc_network {
  properties {
    name = "network"
  }
}

subnet concrete_net {
  properties {
    name = "subnetwork";
  }
}
```

```

        ip_cidr_range => net1;
        region = "us-centrall";
        network = "vpc_network";
    }
    maps net1
  }
}
provider docker {
  cont_image concrete_wp_image {
    properties {
      name = wordpress;
      image = wordpress:5.8.0;
      ports = "8080:80";
      env {
        WORDPRESS_DB_HOST => infra.dbms_vm;
        WORDPRESS_DB_USER => app.database.username;
        WORDPRESS_DB_PASSWORD => app.database.password;
        WORDPRESS_DB_NAME => app.database.name;
      }
    }
  }
  maps container1_image
}
}
}
active con_infra1
}

```

In both concretizations, we define a concrete VM for each VM in the abstract infrastructure, and we link them with the “maps” statement. Provider-specific settings are listed in the “properties” section of each concrete component. For example, in `con_infra1` the WordPress VMs run a VM image containing the Ubuntu Linux operating system, while they run Debian Linux in `con_infra2`. VM instance sizes are also declared in the vendor-specific settings.

In both configurations, the container image providing the WordPress installation is supplied as a Docker image. In its properties, we set some WordPress-specific settings—namely, database credentials—by referring to the values provided in the application layer.

Note that the concrete infrastructures do not provide a VM image for the database VM 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 VMs mapping it.

Finally, the “active” statement sets `con_infra1` as the concrete infrastructure to be used for deployment, while `con_infra2` is disabled.

7.1.2 DOML model with a SaaS database

The second version of the WordPress example uses a SaaS database instead of a custom VM deployment. We describe it by reporting only differences with respect to the first version, described in Section 7.1.1. Its architecture is represented in Figure 23.

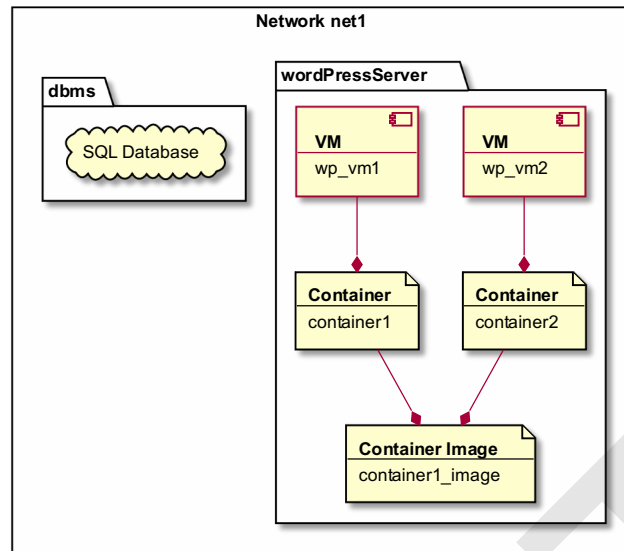


Figure 23. Diagram of the WordPress example with a SaaS database

7.1.2.1 Application Layer

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

```
saas_dbms database {
  provides {
    SQL_interface
  }
  properties {
    identifier      = "education";
    engine          = "postgres";
    engine version  = "13.1";
    username        = "XXX";
    password        = "KKK";
  }
}
```

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

7.1.2.2 Abstract Infrastructure Layer

The only difference in the abstract infrastructure layer is the absence of the “dbms_vm” virtual machine and of its binding with the “database” application in the “deployment” section. Thus, we do not report it again.

7.1.2.3 Concrete Infrastructure Layer

The concretization layer only differs in the DBMS concretizations.

In “con_infra1”, the concrete infrastructure based on AWS, the SaaS database is implemented through Amazon RDS. Hence, the concrete VM called “concrete_dbms_vm” is replaced with the following:

```
saas_dbms concrete_saas_dbms {
  properties {
    instance_class      = "db.t3.micro";
    allocated_storage   = 1;
    publicly_accessible = true;
    skip_final_snapshot = true;
  }
  maps database
}
```


The ICG then automatically maps the database to an Amazon RDS instance, configured with the settings specified in the “properties” section.

In “con_infra2”, we replace the “concrete_dbms_vm” block with

```
saas_dbms concrete_saas_dbms {
  properties {
    name = "saas_db";
    region = "europe-west1";
    tier = "db-f1-micro";
    database_version => database;
  }
  maps database
}
```

This is mapped by the ICG to a Google Cloud Database instance with the specified properties.

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 generate 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 infrastructure consists of the following components:

- A web interface for uploading and downloading images.
- A FaaS image-resizing software program.
- Two storage buckets that store respectively the input and output images.
- A notification service that manages the communication between the web app and the resizing function.

We show its architecture in Figure 24.

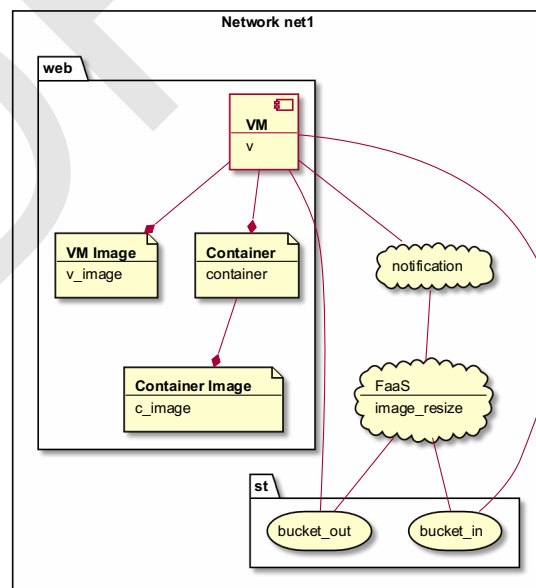


Figure 24. Diagram of the example FaaS thumbnail generator.

Please note that this example is currently at an earlier stage of development than the ones presented in Section 7.1. Thus, it may not yield a fully functional deployment yet.

7.2.1 Application Layer

The application layer is defined as follows:

```
doml faas

application ImageResizeApp {
  software_component image_resize {
    provides {
      handle_image
    }
    consumes {
      storage_interface,
      bucket_in,
      bucket_out
    }
    properties {
      source_code = "https://github.com/xlab-si/xopera-
examples/blob/master/cloud/aws/thumbnail-generator-with-
vm/modules/lambda/function/image_resize.py"
    }
  }
  saas st {
    provides {
      storage_interface
      bucket_in
      bucket_out
    }
  }
  software_component notification {
    consumes {
      handle_image,
      bucket_in
    }
  }
  software_component web {
    consumes {
      storage_interface,
      bucket_out
    }
    properties {
      source_code = "https://github.com/xlab-si/xopera-
examples/tree/master/cloud/aws/thumbnail-generator-with-
vm/modules/ec2_web_app/web_app"
    }
  }
}
```

The image resizing program, the web interface and the notification service are defined as software component. The source files for some of them are supplied as properties. The ICG will build a suitable environment for running during the deployment. Finally, the storage service providing input/output buckets is defined as SaaS.

7.2.2 Abstract Infrastructure Layer

The abstract infrastructure layer is defined below:

```
infrastructure infra {
  vm v {
    ifaces {
      iface i1 {
        address "16.0.0.3"
      }
    }
  }
}
```

```

}
vm_image v_img {
  file "v.iso"
  generates v
}
container c { }
cont_image c_img {
  generates c
}
}
faas f { }
sto s { }
net net1 {
  address "16.0.0.0/24"
  protocol "tcp/ip"
}
autoscale_group g1 { v }
}

```

Here we define a VM that will host the web application. The VM will be generated from a user-supplied VM image. The web application is installed on the VM by deploying a container. The FaaS component, the storage service and the internal network are also defined. No details are specified for the container, the FaaS component and the storage service at this level: they are refined in the concrete infrastructure layer.

The components above are linked to the application-layer services they implement in the following deployment block:

```

deployment config1 {
  c -> v,
  image_resize -> f,
  web -> c,
  st -> s
}
active config1

```

7.2.3 Concrete Infrastructure Layer

The infrastructure is concretized below using AWS as the cloud provider.

```

concretizations {
  concrete_infrastructure con_infra {
    provider aws {
      vm concrete_vm {
        properties {
          instance_type = "t2.micro"
        }
        maps v
      }
      faas concrete_f {
        maps f
      }
      storage concrete_s {
        maps s
      }
      net concrete_net {
        properties {
          cidr_block => net1;
          name = "piacere_subnet";
        }
        maps net1
      }
    }
    provider docker {
      cont_image concrete_c_img {
        properties {
          file "Dockerfile"
        }
      }
    }
  }
}

```

```
        maps c_img
      }
    }
  }
  active con_infra
}
```

Each abstract infrastructure component is mapped to one in the concrete layer.

The ICG generates a deployment by merging the settings specified in the application and abstract infrastructure layers with those added here. In particular, the container image is defined here by selecting Docker as the provider and is built from a user-supplied Docker configuration file.

8 Reflections on the development of the extension mechanism DOML-E

The cloud market is constantly evolving, and thus, all the tools related to cloud applications deployment must evolve alongside it. DOML, as model for application deployment description, must also be able to cope with these changes in a seamless way.

In order to achieve this objective, DOML includes extensions mechanisms that will allow the users to create new cloud related concepts (mainly new abstract and concrete infrastructure concepts) and/or adding new properties to existing ones. These extensions mechanisms are referred to as DOML-E.

Using DOML-E, the DOML can be extended in two different ways:

- *Creation of new concepts.* The DOML includes a set of basic semantic-less concepts that can be used to extend DOML with new concepts. These concepts will require the definition of a metaClassName (i.e., a string that defines the concept behind the new element). Extension elements exist in all the DOML layers, and they can be easily identified, as they use the "Ext" prefix. More specifically, each of the layers of the DOML metamodel include a class extending the ExtensionElement class defined in the Common Layer. For instance, the Application Layer includes the class ExtApplicationComponent that is able to incorporate 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 DOML allows the user to extend the set of properties and attributes associated to one particular DOML concept to further increase its expressiveness. The same mechanism is also usable to fully define extended concepts added to the DOML.

Currently, these two extension mechanisms allow the users to add as many new concepts to DOML as required as the technology evolves, making it easier to keep up with the advancement of the state of the art.

Here we give examples for the DOML extension mechanisms using above approaches by considering a scenario of Internet of Things (IoT). An IoT infrastructure aims at providing a platform to connect large number of devices (e.g., sensors, smart devices, etc.), enabling the communication among the devices, the storage, processing, analyzing of data. To extend DOML with the capability of modeling IoT applications, we may create some fundamental concepts, e.g., an *ExtIoTCore* (like a mobile network core) in the Abstract Infrastructure layer of DOML for handling the connection and management of devices, an *ExtDataflow* service element in Application layer for the streaming or batch analysis of data, etc., while new properties like *Uniquelidentifiers of devices, the corresponding device type, location, etc.* can be introduced.

It is important to note that for these extensions to be fully compatible with PIACERE, the other PIACERE components must be able to correctly exploit the information introduced by these new DOML concepts. For example, if a new infrastructure element is included in a DOML model, new ICG generation templates associated to this concept must be created. The new templates must be created by an expert developer following the guidelines that will be provided in the next ICG release. If the new infrastructure concept is supported by multiple providers, then specific templates for this concept on each provider must be developed. Each new template will have specific placeholders to be substituted, at code generation time, with the values provided in the corresponding DOML concept. On the other hand, if new properties are added to existing concepts, the corresponding ICG templates must also be updated in order to obtain valid code.

The update will include new code and the proper placeholders to be substituted with the value of the new properties. In both cases the ICG Intermediate Representation, used as input by the code generator plugins, will have to be expanded either with the new concept or with the new properties, including the key-value pairs that can provide the needed value for the new properties or for the properties of the new concept.

Further information about the DOML-E mechanisms can also be found in the Annex [1], alongside the specification of the DOML language.

DRAFT

9 Conclusions

This deliverable has presented the current version of the PIACERE DOML and some reflections on the extension mechanism (DOML-E).

The DOML language has been defined around two main principles: 1) a single model for multiple IaC targets and 2) the well-known principle of separation of concerns. Our hypothesis, to be verified in the forthcoming experimentations, is that these principles enhance the usability of the language and its ability to help users in developing multiple configurations from the same software system. The initial examples presented in this deliverable provides a first preliminary confirmation of this hypothesis.

The plan for the next iteration of the DOML includes its validation through the PIACERE use case and, possibly, through other examples. The experiments will allow us to check the completeness of the language and its ability to provide to the ICGs all pieces of information needed for the translation into the target IaCs. Moreover, they will allow us to check the verifiability of the DOML using the model checking-based approach offered by the VT, and the expressiveness of the language for what concerns the optimization features. Based on the outcomes of these experiments, a new more polished version of the language will be created. Particular attention will be posed to the DOML-E and to the ability to incorporate in the DOML all aspects to be modelled in order to make all PIACERE tools as flexible as possible. The results on these activities will be reported in D3.2 at Month 24 and D3.3 and Month 30.

10 References

- [1] PIACERE consortium;, "PIACERE DOML Specification Annex v.0.1," 2021.
- [2] M. Artac, T. Borovšak, E. Di Nitto, M. Guerriero, D. Perez-Palacin and D. A. Tamburri, "Infrastructure-as-Code for Data-Intensive Architectures: A Model-Driven Development Approach," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018.
- [3] C. Ebert, G. Gallardo, J. Hernantes and N. Serrano, "DevOps," *IEEE Software*, vol. 33, no. 3, 2016.
- [4] Puppet Company, "Welcome to Puppet 7.12.1," [Online]. Available: https://puppet.com/docs/puppet/7/puppet_index.html. [Accessed December 2021].
- [5] Chef, "Chef Infra: Powerful Policy-Based Configuration Management System Software," [Online]. Available: <https://www.chef.io/products/chef-infra>. [Accessed December 2021].
- [6] NovelVista, "What is Puppet and What are its Key Components," July 2021. [Online]. Available: <https://www.novelvista.com/blogs/devops/what-is-puppet-and-what-are-key-components>. [Accessed December 2021].
- [7] Red Hat Ansible, "Ansible documentation".
- [8] M. Rutkowski, C. Lauwers, C. Noshpitz and C. Curescu, "TOSCA Simple Profile in YAML Version 1.3," OASIS, 2020.
- [9] Cloudify, "What is TOSCA?," [Online]. Available: <https://cloudify.co/what-is-tosca/>. [Accessed December 2021].
- [10] OASIS, "Topology and Orchestration Specification for Cloud Applications Version 1.0," 2013. [Online]. Available: <http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html>.
- [11] XLAB, "XOPERA," [Online]. Available: <https://xlab-si.github.io/xopera-docs/introduction.html>. [Accessed December 2021].
- [12] Hashi Corp., "Terraform," [Online]. Available: <https://www.terraform.io/>. [Accessed December 2021].
- [13] freeCodeCamp, "Terraform Workflow: How to Work Individually and in a Team," [Online]. Available: <https://www.freecodecamp.org/news/terraform-workflow-working-individually-and-in-a-team/>.
- [14] Amazon Web Services, "AWS CloudFormation - Speed up cloud provisioning with infrastructure as code," [Online]. Available: https://aws.amazon.com/cloudformation/?nc1=h_ls. [Accessed December 2021].
- [15] Amazon, "AWS CloudFormation Features," [Online]. Available: https://aws.amazon.com/cloudformation/features/?nc1=h_ls.

- [16] Reply, "AWS CloudFormation," [Online]. Available: <https://www.reply.com/en/content/aws-cloudformation>.
- [17] I. Miell and A. Hobson Sayers, *Docker in practice*, second edition, Shelter Island: Manning, 2019.
- [18] B. Jšrg, J. Lappalainen and K. Kastrantas, "Modeling the semantics of contextual and content-specific research metadata using ontology languages: issues on combining CERIF and OWL," *Procedia Computer Science*, vol. 9, pp. 1563-1570, 2012.
- [19] Canonical Ltd, «Container and virtualization tools,» [En línea]. Available: <https://linuxcontainers.org/>. [Último acceso: 23 12 2021].
- [20] Kubernetes, "What is Kubernetes?," [Online]. Available: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes>. [Accessed 2021].
- [21] kubernetes, "Secrets," [Online]. Available: <https://v1-20.docs.kubernetes.io/docs/concepts/configuration/secret/>.
- [22] Kubernetes,, "Kubernetes Components," [Online]. Available: <https://kubernetes.io/docs/concepts/overview/components/>.
- [23] G. Blair, N. Bencomo and R. B. France, "Models@run.time," *Computer*, vol. 42, no. 10, p. 22–27, 2009.
- [24] E. Di Nitto et al., "Supporting the Development and Operation of Multi-cloud Applications: The MODAClouds Approach," in *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2013.
- [25] K. Kritikos, J. Domaschka and A. Rossini, "SRL: A scalability rule language for multi-cloud environments," in *IEEE 6th International Conference on Cloud Computing Technology and Science*, 2014.
- [26] G. Casale et al., *Practical DevOps for Big Data*, WikiBooks, 2018.
- [27] G. e. a. Casale, *Practical DevOps for Big Data*, WikiBooks, 2018.
- [28] SODALITE Consortium, "Project SODALITE," [Online]. Available: <https://sodalite.eu/>. [Accessed December 2021].
- [29] E. Di Nitto et al., "An Approach to Support Automated Deployment of Applications on Heterogeneous Cloud-HPC Infrastructures," in *2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2020.
- [30] DECIDE Consortium, "DECIDE FactSheet," 2017.
- [31] DECIDE Consortium,, "DECIDE D2.5 Detailed architecture," 2018.
- [32] DECIDE Consortium, "DECIDE D3.12 Final multi-cloud native application controller," 2019.

- [33] G. Casale et al., "RADON: Rational decomposition and orchestration for serverless computing," *SICS Software-Intensive Cyber-Physical Systems*, vol. 35, no. 1, pp. 77-87, 2021.
- [34] RADON Consortium, "RADON: Open Source DevOps for Serverless Applications," [Online]. Available: <https://radon-h2020.eu/wp-content/uploads/2021/07/RADON-Booklet.pdf>. [Accessed December 2021].
- [35] B. Selic, "The pragmatics of model-driven development.," *IEEE Software*, vol. 20, no. 5, pp. 19-25, 2003.
- [36] F. J. Budinsky, M. A. Finnie, J. M. Vlissides and P. S. Yu, "Automatic code generation from design patterns," *IBM Systems Journal*, vol. 35, no. 2, pp. 151-171, 1996.
- [37] L. Geiger, C. Schneider and C. Reckord, "Template-and modelbased code generation for MDA-Tools," 2005.
- [38] Apache Software Foundation, "The Apache Velocity Project," [Online]. Available: <https://velocity.apache.org/>.
- [39] Fujaba Team, [Online]. Available: <https://github.com/fujaba>.
- [40] Z. Chared and S. S. Tyszberowicz, "Projective Template-Based Code Generation," *In CAiSE Forum*, pp. 81-87, 2013.
- [41] Codiscent, [Online]. Available: <http://codiscent.com/>. [Accessed December 2021].
- [42] E. Syriani, L. Luhunu and H. Sahraoui, "Systematic mapping study of template-based code generation," *Computer Languages, Systems & Structures*, vol. 52, pp. 43-62, 2018.
- [43] L. Luhunu and E. Syriani, "Comparison of the expressiveness and performance of template-based code generation tools," 2017.
- [44] M. Bork, L. Geiger, C. Schneider and A. Zündorf, "Towards roundtrip engineering-a template-based reverse engineering approach," in *European Conference on Model Driven Architecture-Foundations and Applications*, 2008.
- [45] P. Stevens, "A landscape of bidirectional model transformations," in *International Summer School on Generative and Transformational Techniques in Software Engineering*, 2007.
- [46] A. Königs, "Model transformation with triple graph grammars," in *Model Transformations in Practice Satellite Workshop of MODELS*, 2005.
- [47] J. Sandobalin, E. Insfran and S. Abrahao, "An infrastructure modelling tool for cloud provisioning," in *2017 IEEE International Conference on Services Computing (SCC)*, 2017.
- [48] J. Sandobalin, E. Insfran and S. Abrahao, "An Infrastructure Modeling Approach for Multi-Cloud Provisioning," in *Information Systems Development: Designing Digitalization (ISD2018 Proceedings)*, Lund, Sweden, 2018.

- [49] F. Jouault, F. Allilaire, J. Bézivin and I. Kurtev, "ATL: A model transformation tool," *Science of computer programming*, vol. 72, no. 1-2, pp. 31-39, 2008.
- [50] Eclipse Foundation, "Acceleo," [Online]. Available: <https://www.eclipse.org/acceleo/>. [Accessed December 2021].
- [51] J. Sandobalin, E. Insfran and S. Abrahao, "On the Effectiveness of Tools to Support Infrastructure as Code: Model-Driven Versus Code-Centric," *IEEE*, vol. 8, pp. 17734-17761, 2020.
- [52] PIACERE Consortium;, "D2.1 - PIACERE DevSecOps Framework Requirements specification, architecture and integration strategy - v1".
- [53] A. De Carvalho et. al., Performance Comparison of Terraform and Cloudify as Multicloud Orchestrators, 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), 2020, pp. 380-389.
- [54] Eclipse Foundation, "Eclipse Modeling Framework Technology (EMFT)," [Online]. Available: <https://www.eclipse.org/modeling/emft/?project=emfatic>. [Accessed December 2021].



PIACERE

Annex to D3.1

DOML Specification

Editor(s):	Adrian Noguero
Responsible Partner:	Go4It
Status-Version:	V0.1
Date:	20.12.2021
Distribution level (CO, PU):	Public

Project Number:	101000162
Project Title:	PIACERE

Title of Deliverable:	Annex to D1.3
Due Date of Delivery to the EC	30.11.2021

Workpackage responsible for the Deliverable:	WP3 - Plan and create Infrastructure as Code
Editor(s):	Go4It
Contributor(s):	Go4It, Polimi
Reviewer(s):	Alfonso de la Fuente (Prodevelop)
Approved by:	All Partners
Recommended/mandatory readers:	WP4, WP5, WP6, WP7

Abstract:	This annex is accompanying document to Deliverable D3.1 - PIACERE Abstractions, DOML and DOML-E - v1. It include the detailed specification of the DOML concepts. It will be updated periodically to account for the development of the language.
Keyword List:	DOML, Modelling abstractions
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
v0.01	08.10.2021	First draft version	GO4IT
v0.02	10.12.2021	Completed the specification v1.0 of DOML. Added the concrete layer and all DOML-E mechanisms. Completed the properties and updated examples	GO4IT
V0.1	20.12.2021	Syntax definition added, revision of the whole content	Polimi

Table of contents

Terms and abbreviations.....	7
Executive Summary.....	8
1 Description of DOML.....	9
1.1 DOML Layers	9
2 Commons Layer.....	11
2.1 DOMLElement Class (abstract).....	11
2.2 Property Class.....	12
2.3 DOMLModel Class	12
2.4 Configuration Class.....	13
2.5 Deployment Class.....	14
2.6 ExtensionElement Class (abstract)	14
2.7 Requirement Class.....	14
2.8 RangedRequirement Class	15
2.9 EnumeratedRequirement Class.....	16
2.10 DeploymentRequirement Class (abstract)	16
2.11 DeploymentToNodeTypeRequirement Class	17
2.12 DeploymentToNodeWithPropertyRequirement Class	17
2.13 DeploymentToSpecificNodeRequirement Class.....	18
3 Application Layer.....	19
3.1 ApplicationLayer Class.....	19
3.2 ApplicationComponent Class (abstract)	20
3.3 SoftwarePackage Class.....	20
3.4 SaaS Class	21
3.5 SoftwareInterface Class.....	21
3.6 DBMS Class.....	22
3.7 SaaSDBMS Class	22
3.8 ExtApplicationComponent Class	22
4 Infrastructure Layer	23
4.1 InfrastructureLayer Class.....	23
4.2 InfrastructureElement Class (abstract)	24
4.3 ComputingNode Class (abstract).....	24
4.4 PhysicalComputingNode Class	25
4.5 ComputingNodeGenerator Class (abstract)	25
4.6 VirtualMachine Class.....	26
4.7 Location Class.....	26
4.8 Container Class.....	26

4.9	GeneratorKind Enum.....	27
4.10	ComputingNodeGenerator Class (abstract)	27
4.11	VMIImage Class	27
4.12	ContainerImage Class	28
4.13	AutoScalingGroup Class.....	28
4.14	Storage Class	29
4.15	FunctionAsAService Class	29
4.16	ExtInfrastructureElement Class (abstract)	29
4.17	Network Class.....	30
4.18	Subnet Class	31
4.19	NetworkInterface Class	31
4.20	Firewall Class	31
4.21	RuntimeOrchestrationEnvironment Class.....	32
5	Concrete Layer	33
5.1	ConcreteInfrastructure Class.....	33
5.2	ConcreteElement Class (abstract)	34
5.3	RuntimeProvider Class	34
5.4	VirtualMachine Class.....	35
5.5	Network Class.....	35
5.6	Storage Class	35
5.7	FunctionAsAService Class	36
5.8	AutoScalingGroup Class.....	36
5.9	ExtConcreteElement Class.....	36
6	Optimization Layer	38
6.1	OptimizationLayer Class	38
6.2	OptimizationObjective Class (abstract)	39
6.3	CountObjective Class.....	39
6.4	MeasurableObjective Class	39
6.5	OptimizationSolution Class	40
6.6	ExtOptimizationObjective Class (abstract)	40
7	DOML Text Syntax	41
8	DOML Examples	45
8.1	Simple Web Application	45
8.2	Optimization Problem Example.....	47
9	Conclusions	49

List of figures

FIGURE 1. COMMONS LAYER DIAGRAM (EXCLUDING REQUIREMENT SUBCLASSES).....	11
FIGURE 2. COMMONS LAYER REQUIREMENTS DIAGRAM.....	15
FIGURE 3 APPLICATION LAYER DIAGRAM.....	19
FIGURE 4. INFRASTRUCTURE LAYER DIAGRAM SHOWING INFRASTRUCTURE NODES	23
FIGURE 5. INFRASTRUCTURE LAYER DIAGRAM SHOWING NETWORK RELATED CONCEPTS	30
FIGURE 6. INFRASTRUCTURE LAYER DIAGRAM	33
FIGURE 7. OPTIMIZATION LAYER DIAGRAM	38

DRAFT

Terms and abbreviations

CSP	Cloud Service Provider
DevOps	Development and Operation
DoA	Description of Action
EC	European Commission
GA	Grant Agreement to the project
IaC	Infrastructure as Code
IEP	IaC execution platform
IOP	IaC Optimization
KPI	Key Performance Indicator
SW	Software

Executive Summary

This document contains the main description of the DOML language specification, as well as its extension mechanisms (DOML-E). The goal of the document is to serve as cornerstone for the implementation of DOML based solutions in PIACERE, ranging from the IDE to the optimization algorithms.

DRAFT

1 Description of DOML

DOML specifies a common language for addressing the definition, deployment and operation of complex cloud-based applications inside the PIACERE framework. DOML is intended to be used by users with different degrees of expertise, therefore, it has been conceived to be easy to use by non-expert users, but also expressive enough to allow expert users to get the most out of it.

DOML is a declarative language, thus, each of the layers describe what the application and infrastructure should look like after all the deploying is done. However, DOML allows the user to integrate imperative scripts, to actually describe some specific configurations whenever needed. The main goal of DOML is to serve as a bridge to the many IaC languages that currently exist (e.g. Terraform, TOSCA, Ansible...), providing a degree of expressiveness that allows the PIACERE framework to generate IaC code in those mentioned formats easily.

In addition, DOML is intended to be used with the PIACERE optimization mechanisms. To achieve this DOML allows the user to define different application deployment configurations, as well as different infrastructure configurations, and it includes a specific layer to define optimization objectives and constraints.

Finally, DOML is envisaged as an evolving entity capable of coping with the constant advancements in the cloud computing state-of-the-art. As such, DOML includes extension mechanisms built inside that allow the user and the tools using DOML to create new concepts for any of the layers in DOML, as well as extending existing ones with new properties and attributes. This extension mechanisms are collectively called DOML-E.

This specification is intended to be used as a reference for the implementation of all DOML related tools and guides.

1.1 DOML Layers

The DOML language specification is split into several packages, referred to as “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 build up for a comprehensive application description.

The **Commons Layer** contains the main abstract application agnostic concepts that are shared among different layers. 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.

The **Application Layer** 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.

The **Infrastructure Layer** 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 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.

The **Concrete Layer** 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.

The **Optimization Layer** 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.

DRAFT

2 Commons Layer

The following diagram shows the main elements of the Commons Layer in DOML:

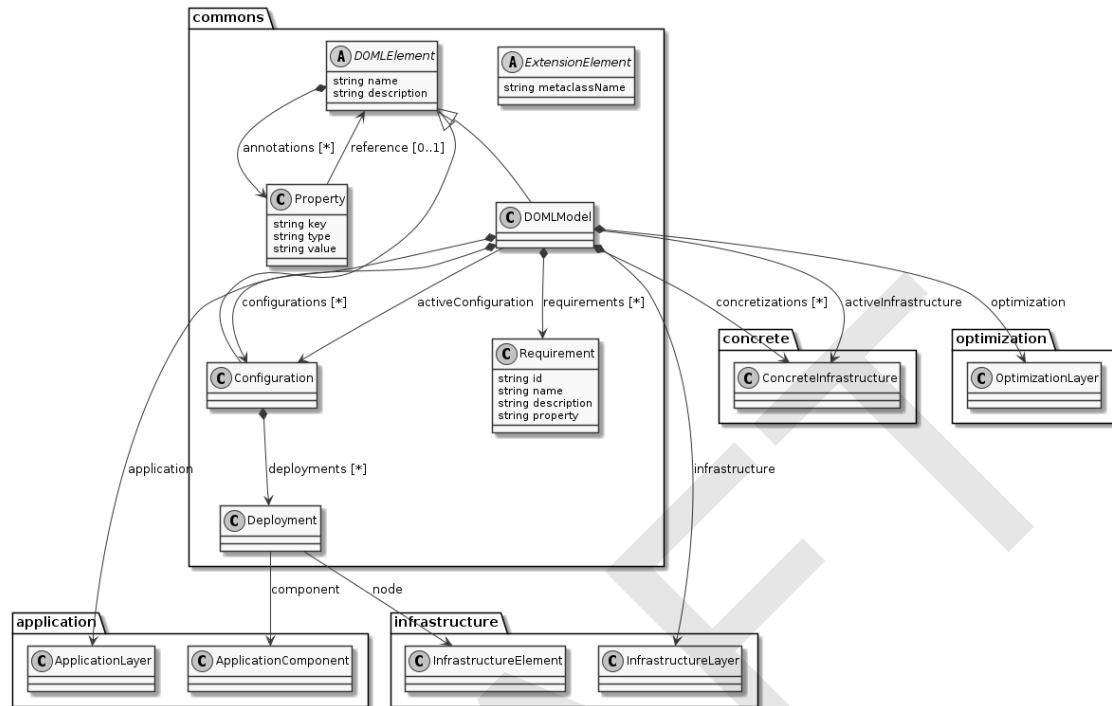


Figure 1. Commons Layer diagram (excluding Requirement subclasses)

2.1 DOMLElement Class (abstract)

A DOMLElement represents any element inside the DOML language and it is intended to be the top meta-element of the DOML.

Attributes

name: String [1]	An identifier for this DOML Element
description: String [0..1]	An optional textual description of the Element. Used for documenting the element or similar purposes.

Associations

annotations: Property [0..*]	A set of properties used to modify the semantics of this particular element. These properties will add to the final semantics of the element, refining the DOML element to which they are applied. These properties will serve as the main extension mechanisms for DOML.
contributesTo: Requirement [0..*]	The set of requirements this DOMLElement contributes to achieving. This association is derived from the predicatesOn association of the Requirement class.

Constraints

* All properties added to a DOML element must have different keys.

Usage

DOMLElement is the common parent of all elements in DOML but the Property class. It is also the enabler for DOML-E extensions through the use of Property elements.

2.2 Property Class

A Property represents an additional information added to any DOML Element to further refine their meaning or semantics.

Attributes

key: String [1]	An identifier for this Property
type: String [0..1]	An optional name defining the data type for this property
value: String [0..1]	An optional textual information associated with this Property instance.

Associations

reference: DOMLElement [0..1]	An optional link to another DOML Element relevant for this property.
-------------------------------	--

Constraints

* All properties owned by a DOML element must have different keys.

Usage

Instances of the Property class are used to add information to DOML elements that cannot be described using that element's attributes and associations. Properties can be used as any or both attributes and associations to refine any DOML element.

2.3 DOMLModel Class

A DOMLModel represents the design and development space for a cloud application or set of applications. The DOML model provides access to the different points of view of this space through the use of the model layers.

Superclass

DOMLElement

Associations

application: ApplicationLayer [0..1]	A reference to the Application Layer instance associated with this model.
infrastructure: InfrastructureLayer [0..1]	A reference to the Concrete Infrastructure Layer instance associated with this model.

concretizations: ConcreteInfrastructure [0..*]	A list of concrete infrastructures that map on to abstract infrastructure layer elements.
activeInfrastructure: ConcreteInfrastructure [0..1]	The ConcreteInfrastructure considered active for the current DOML specification
optimization: OptimizationLayer [0..1]	A reference to the Optimization Layer instance associated with this model.
configurations: Configuration [0..*]	All possible configurations of the current DOML specification
activeConfiguration: Configuration [0..1]	The Configuration instance considered as active
requirements: Requirement [0..*]	The set of requirements that are applicable to the current DOML specification.

Usage

A DOML model is intended to be used as the container for all the DOML layers defined in a particular design space. Each of those layers will provide a different point of view of the design space of the application. This element should be used as the root element of any DOML model.

2.4 Configuration Class

A Configuration describes how an application is intended to be deployed on top of the cloud infrastructure, and what credentials, parameters, etc. will apply to each of the application and/or infrastructure elements.

Superclass

DOMLElement

Associations

deployments: [0..*]	Deployment	A set of Deployment instances describing each of the links between an application component and a node of the infrastructure.
---------------------	------------	---

Constraints

* There must not be two Deployment instances inside a Configuration element with the same source and target elements.

Usage

A configuration must fully describe how an application will operate on top of a particular infrastructure. Since the parameters associated to each DOML element, whether it is an application component or an infrastructure node, will differ, the configuration element will use the Property list to include them, using the reference Association of the Property to describe which model element that particular parameter affects to.

2.5 Deployment Class

A Deployment element describes an association between an application component (e.g. a web application) and the infrastructure element that will host it (e.g. a Virtual Machine).

Associations

source: ApplicationComponent [1]	The application component that will be deployed.
target: InfrastructureElement [1]	The infrastructure element that will host/support the application component.

Usage

The deployment is designed to establish 1 to 1 relationships between application and infrastructure elements.

2.6 ExtensionElement Class (abstract)

A ExtensionElement abstract metaclass is used as the common meta-type for all the classes that are part of DOML-E extension mechanisms.

Attributes

metaclassName: String [1]	The name of the metaclass that will be added to DOML by using the extension class instance.
---------------------------	---

Usage

The extension element class must never be instantiated nor subclassed. Instead all extension metaclasses in DOML (i.e. ExtApplicationComponent, ExtInfrastructureElement, ExtConcreteElement and ExtOptimizationObjective) extend this metaclass, in addition to other extensions.

2.7 Requirement Class

A Requirement represents an objective to be achieved by the current DOML specification. Requirements, whether they are functional, non-functional or optimization objectives, must be described in plain text and also annotations can be used to further qualify it, if needed.

Attributes

identifier: String [1]	A unique identifier for this requirement.
title: String [0..1]	An optional meaningful title for the requirement.
description: String [0..1]	A text further specifying the requirement.
property: String [0..1]	The property of the DOMLElement instances this requirement predicates on.

Associations

predicatesOn: DOMLElement A reference to the set of DOMLElement instances this requirement predicates on.

Constraints

* All requirements in a DOML model must have different identifiers.

Usage

Requirements are used to model objectives and restrictions the current DOML design must meet. These objectives should be as formal as possible; however, they can also be used in a less formal way using the textual attributes. The way to define them in a formal way is by using the “property” and the “predicatesOn” members. The Requirement class is also the parent of all formal requirements defined in DOML. The following diagram shows the requirements section of the commons layer in DOML.

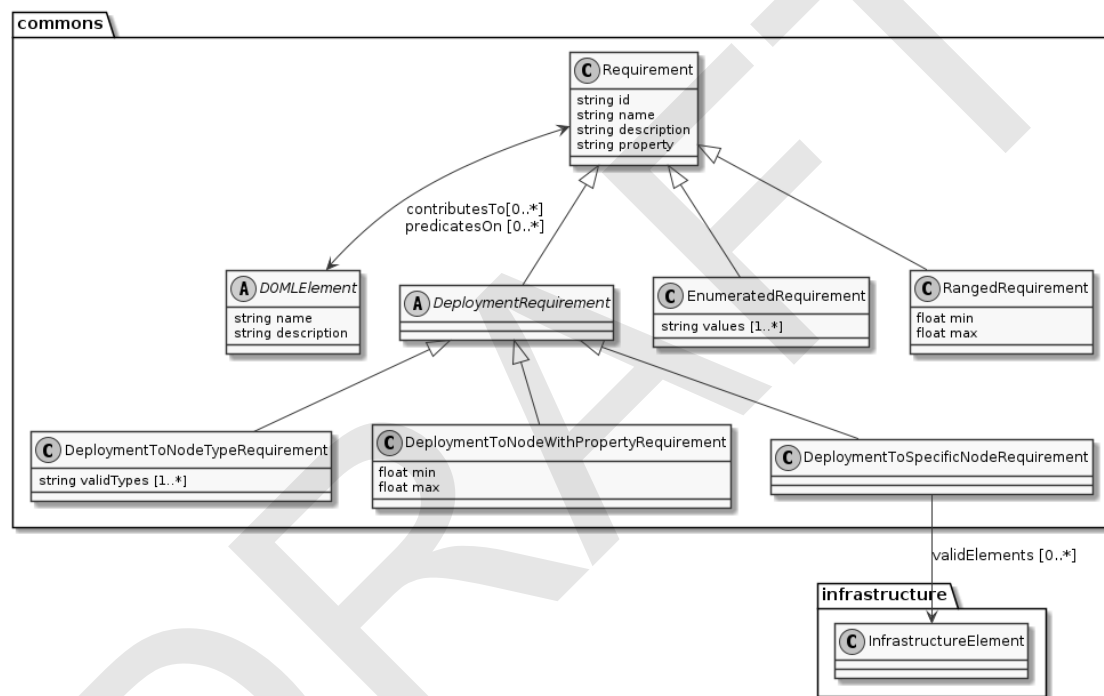


Figure 2. Commons Layer Requirements diagram

2.8 RangedRequirement Class

A RangedRequirement is a formal requirement instance which establishes a range of valid values to a property in a set of DOMLElements.

Superclass

Requirement

Attributes

min: Float [0..1]	The minimum value of the property.
max: Float [0..1]	The maximum value of the property.

Constraints

- * The property attribute of a RangedRequirement must always be set.
- * The predicatesOn association must always be linked to at least one DOMLElement for a RangedRequirement
- * At least the max or the min attributes of a RangedRequirement must be set.
- * A ranged requirement can only be applied to numeric properties.

Usage

A ranged requirement should be used to establish limits to the numeric properties that need them.

2.9 EnumeratedRequirement Class

A EnumeratedRequirement describes a formal requirement that restricts the number of valid values a property of a certain DOML element may take.

Superclass

Requirement

Attributes

values: String [1..*]

The set of values that are valid for the property referred by this requirement.

Constraints

- * The property attribute of a EnumeratedRequirement must always be set.
- * The predicatesOn association must always be linked to at least one DOMLElement for a EnumeratedRequirement
- * At least one value must be set in the values attribute.

Usage

An enumerated requirement is used to set a list of valid values for a particular property.

2.10 DeploymentRequirement Class (abstract)

A DeploymentRequirement class describes a restriction to be applied to the definition of configurations in the current DOML.

Superclass

Requirement

Constraints

- * The predicatesOn association must always be linked to at least one DOMLElement for a DeploymentRequirement and they must all be ApplicationComponent instances.

Usage

A DeploymentRequirement is used as the common parent class to all deployment related formal requirements in DOML.

2.11 DeploymentToNodeTypeRequirement Class

A DeploymentToNodeTypeRequirement describes a formal requirement that restricts types of infrastructure elements an application component can be deployed to.

Superclass

DeploymentRequirement

Attributes

validTypes: String [1..*]	The set of valid meta-types the application components this requirement predicates on can be deployed to.
---------------------------	---

Constraints

- * At least one value must be set in the validTypes attribute.
- * Values in validTypes must all be valid names of meta-classes in DOML infrastructure layer that extend the InfrastructureElement class.

Usage

A requirement of this kind is used to make an application component or a set of components deployable only into certain types of infrastructure elements (for example, make a software package only deployable to physical nodes).

2.12 DeploymentToNodeWithPropertyRequirement Class

A DeploymentToNodeWithPropertyRequirement describes a formal requirement that restricts the infrastructure elements an application component can be deployed to according to the value of a property.

Superclass

DeploymentRequirement

Attributes

min: Float [0..1]	The minimum value of the property.
max: Float [0..1]	The maximum value of the property.
values: String [0..*]	The set of values that are valid for the property referred by this requirement.

Constraints

- * The property attribute of a DeploymentToNodeWithPropertyRequirement must always be set.
- * At least the max, the min or the values attributes of a requirement of this kind must be set.

- * If values is not empty, then min and max cannot be set.
- * If min and/or max are set, then values has to be empty.

Usage

A `DeploymentToNodeWithPropertyRequirement` is used to restrict the valid infrastructure nodes an application component can be deployed to according to the value of a property of the target infrastructure element (for example, a software interface can only be attached to a network interface with a minimum speed of 1Gbps, or a dbms component can only be deployed to a node with location equal to Europe).

2.13 DeploymentToSpecificNodeRequirement Class

A `DeploymentToSpecificNodeRequirement` describes a formal requirement that restricts the set of valid infrastructure element an application component can be deployed to a specific list.

Superclass

`DeploymentRequirement`

Associations

<code>validElements:</code> <code>InfrastructureElement [1..*]</code>	The set of elements the application component referred to by this requirement can be deployed to.
--	---

Usage

A `DeploymentToSpecificNodeRequirement` is used provide a valid set of infrastructure elements to be used to deploy an application component or a set of application components.

3 Application Layer

The following diagram shows the main elements of the Application Layer in DOML:

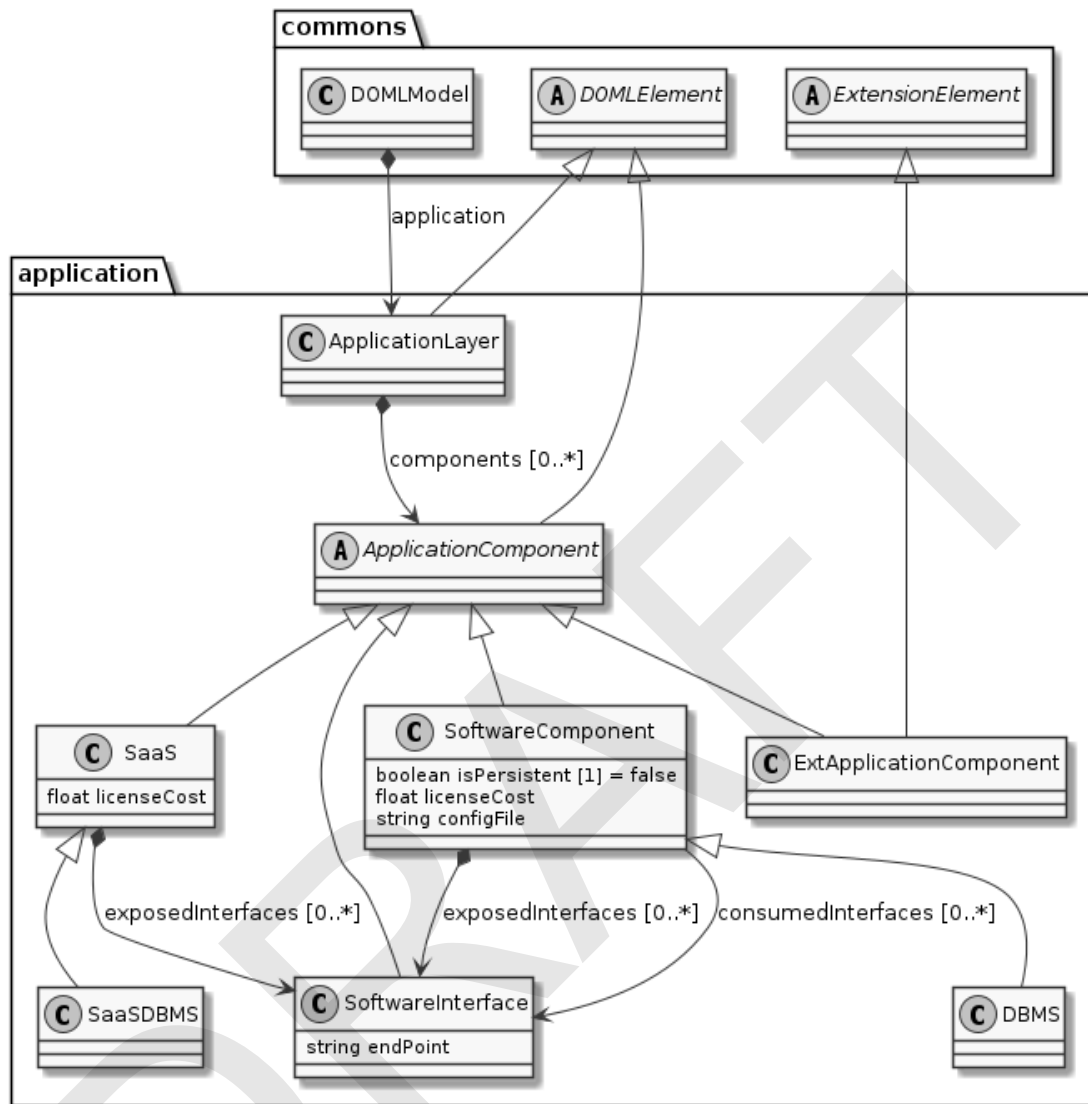


Figure 3 Application Layer diagram

3.1 ApplicationLayer Class

The Application class represent the container for all the components of the application in a DOML design. It is the representation of the Application Layer, and all the functional elements of the cloud application to be deployed must be defined as application components inside it.

Superclass

DOMLElement

Associations

components:
ApplicationComponent [0..*]

A containment reference to all the application components that will be part of the current application layer.

Usage

The Application is designed to be a container for ApplicationComponent instances.

3.2 ApplicationComponent Class (abstract)

The ApplicationComponent describes anything meaningful to the application being deployed in DOML from the functional perspective (e.g. software components, services or APIs). Each application component is susceptible of being deployed to an infrastructure element in the infrastructure model.

Superclass

DOMLElement

Usage

The ApplicationComponent class is intended to be the common parent class for all elements in the application layer. Any common properties must always be specified on this class.

3.3 SoftwarePackage Class

The SoftwarePackage class describes any of the functional software components that conform an application in DOML. A software component may use or provide software interfaces, creating, this way, links among components, APIs and other functional elements in the application layer.

Superclass

ApplicationComponent

Attributes

isPersistent: Boolean [1]	A flag to indicate whether this component persists any information/state during operation. By default the value of this property is <i>false</i> .
licenseCost: Float [0..1]	An optional license cost (in Euro) associated to this software component.
configFile: String [0..1]	The path to the installation and configuration script (e.g. Ansible, Terraform, Shell...) for this software component. This information will be used by IaC generators.

Associations

exposedInterfaces: SoftwareInterface [0..*]	A set of software interfaces provided by this component for other software components to use.
consumedInterfaces: SoftwareInterface [0..*]	The set of software interfaces required by this component to fulfil its role.

Constraints

* Consumed interfaces must always refer to software interfaces exposed by other components or SaaS instances.

Usage

The SoftwarePackage class is intended to describe the main functional components or an application (e.g. web server, a REST API, etc.). It is important to note that software packages should be part of the components to be deployed in and are susceptible of having requirements attached to them.

3.4 SaaS Class

The SaaS class models an API that is external to our application, but relevant for functional purposes.

Superclass

ApplicationComponent

Attributes

licenseCost: Float [0..1]	An optional license cost (in Euro) associated to this SaaS.
---------------------------	---

Associations

exposedInterfaces: SoftwareInterface [0..*]	A set of software interfaces provided by this component for other software components to use.
--	---

Usage

The SaaS class is intended to describe APIs that are external to the current application, but are used by the software components inside it. SaaS components must not have requirements associated to them, the user has no control over them. SaaS instances may, however, define properties related to expected performance, response time, etc. if those are relevant for the current DOML model.

3.5 SoftwareInterface Class

The SoftwareInterface class models a software interface (e.g. a REST API, a TCP/IP connection, etc.) that connects two different application components in the application model.

Superclass

ApplicationComponent

Attributes

endPoint: String	The IP address / hostname / URL through which the service is accessed
------------------	---

Constraints

* A software interface must always be provided by one application component and used by at least one application component in the DOML model.

Usage

The SoftwareInterface class is intended to describe a connector between two different application components.

3.6 DBMS Class

The DBMS describes a software component that includes a Data Base Management System.

Superclass

SoftwarePackage

Constraints

* The isPersistent attribute of a DBMS component must always be set to *true*.

Usage

The DBMS is just a convenient subclass of the more generic SoftwarePackage class to model specifically DBMS.

3.7 SaaSDBMS Class

The SaaSDBMS describes an external API that will provide the DataBase Management System Functionality.

Superclass

SaaS

Usage

The SaaSDBMS class is just a convenient subclass of the more generic SaaS class to model specifically a DBMS providing SaaS.

3.8 ExtApplicationComponent Class

The ExtApplicationComponent describes an instance of a new application layer concept. This class is part of DOML-E extension mechanisms.

Superclasses

ApplicationComponent, ExtensionElement

Usage

The ExtApplicationComponent class is should be used to create instances of concepts and metaclasses not currently available in DOML.

4 Infrastructure Layer

The infrastructure layer describes the abstract infrastructure elements that will be supporting the execution of the application described in the ApplicationLayer. It is important to note that this abstract representation of the infrastructure is intended to be reused, mapping the elements on this layer to concrete instances in the infrastructure (e.g. an abstract virtual machine described in this layer will be mapped to a concrete VM instance, provided by a specific runtime provider, such as AWS or GoogleCloud).

The following diagram shows the main elements of the Infrastructure Layer in DOML related to infrastructure nodes:

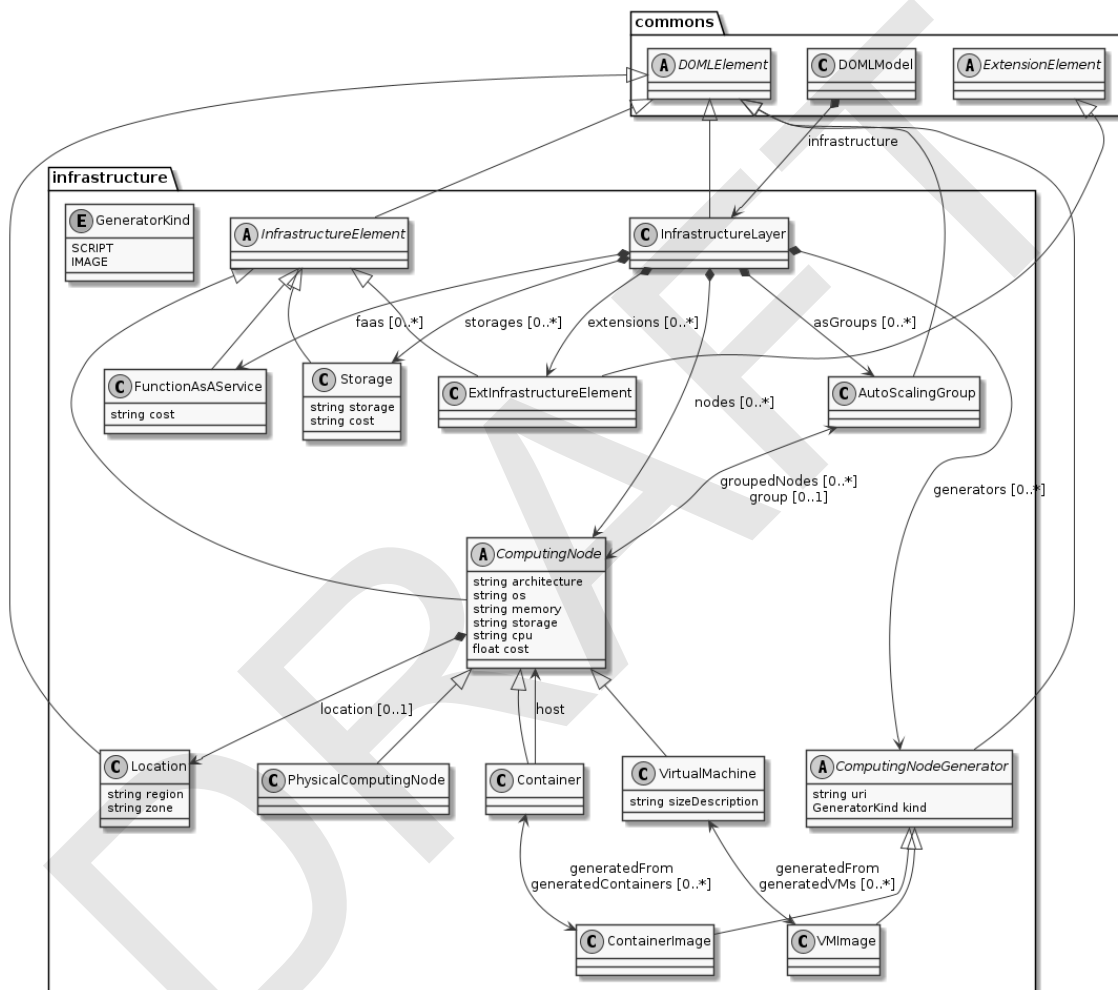


Figure 4. Infrastructure Layer diagram showing infrastructure nodes

4.1 InfrastructureLayer Class

The InfrastructureLayer class is the container for the catalog of infrastructure elements that will be available to the current DOML model.

Superclass

DOMLElement

Associations

providers: RuntimeProvider [0..*]	The list of runtime providers available in the catalogue
nodes: ComputingNode [0..*]	The list of independent computing nodes (not attached to a provider) available in the catalogue
generators: ComputingNodeGenerator [0..*]	The list of virtual machine and container images available in the catalogue
asGroups: AutoScalingGroup [0..*]	The list of independent auto scaling groups (not attached to a runtime provider) available in the catalogue
networks: Network [0..*]	The list of independent networks (not attached to a runtime provider) available in the catalogue
firewalls: Firewall [0..*]	The list of independent firewalls available in the catalogue
orchestrator: RuntimeOrchestrationEnvironment [0..1]	An optional orchestration environment available in the catalogue.
storages: Storage [0..*]	The list of storage resources that will be part of this abstract infrastructure model
faas: FunctionAsAService [0..*]	The list of faas services part of this DOML model infrastructure.

Usage

The InfrastructureLayer is a container element, used as the catalog of infrastructure elements available to deploy the final cloud application using DOML.

4.2 InfrastructureElement Class (abstract)

The InfrastructureElement class represents all infrastructure elements that can have an application component deployed to them.

Superclass

DOMLElement

Usage

The InfrastructureElement is intended to be used as the parent for more concrete elements of the infrastructure model.

4.3 ComputingNode Class (abstract)

The ComputingNode class represents any element that can be used for computing, from a dedicated host to an IoT node.

Superclass

InfrastructureElement

Attributes

architecture: String [0..1]	A string describing the internal architecture of the computing node (e.g. x86, x64, etc.).
os: String [0..1]	A string describing the operating system of this node (e.g. Windows 10, Ubuntu 20.04, etc.).
memory: String [0..1]	A string describing the total memory of this node (e.g. 4GB).
storage: String [0..1]	A string describing the total storage available in this node (e.g. 10TB).
cpu: String [0..1]	A string describing the CPU of the computing node.
cost: Float [0..1]	An optional cost value (in Euro).

Associations

group: AutoScalingGroup [0..1]	Derived property. A link to the group that owns this computing node.
ifaces: NetworkInterface [0..*]	The network interfaces owned by this computing node.
location: Location [0..1]	An optional location for this infrastructure element.

Usage

The CopmutingNode class is intended to be the common parent for all the infrastructure elements capable of executing code.

4.4 PhysicalComputingNode Class

The PhysicalComputingNode class represents a dedicated physical server.

Superclass

ComputingNode

4.5 ComputingNodeGenerator Class (abstract)

The ComputingNodeGenerator class represents all infrastructure elements that describe a virtual computing node.

Superclass

DOMLModel

Usage

The ComputingNodeGenerator is intended to be used as the common parent for all elements defining the characteristics of virtual computing nodes. Often these generators rely on a file which defines them.

Usage

The `PhysicalComputingNode` is used to describe physical computing nodes available for the owner of a cloud application that are going to be used as part of the cloud deployment.

4.6 VirtualMachine Class

The `VirtualMachine` class represents a virtual computing node running on top of a supervisor software.

Superclass

`ComputingNode`

Attributes

`sizeDescription`: String [0..1] An optional string describing the size of the VM.

Associations

`generatedFrom`: `VMImage` [0..1] The image used to generate this virtual machine.

`location`: `Location` [0..1] An optional `Location` object to represent where the VM should be located

Usage

The `VirtualMachine` is used to describe virtual computing nodes running on a supervisor software. In order to be automatically configurable, the virtual machine must define the image that will generate it.

4.7 Location Class

The `Location` class represents the place where a computing node should be.

Superclass

`DOMLModel`

Attributes

`region`: String [1] A string describing the region for this location.

`zone`: String [0..1] An optional attribute to refine the location if the region is not precise enough.

Usage

The `Location` is intended to describe the location of infrastructure elements, more concretely virtual machines and physical machines.

4.8 Container Class

The `Container` class represents a virtual computing node running on top of another computing node.

Superclass

ComputingNode

Associations

generatedFrom: ContainerImage [0..1]	The image used to generate this container.
host: ComputingNode [1]	The computing node that will be the host of this container.

Usage

The Container is used to describe virtual computing nodes, such as Docker containers.

4.9 GeneratorKind Enum

The GeneratorKind enumeration describes the different computing node generation kinds.

Values

SCRIPT, IMAGE

4.10 ComputingNodeGenerator Class (abstract)

The ComputingNodeGenerator class represents all infrastructure elements that describe a virtual computing node.

Superclass

DOMLModel

Attributes

uri: String [0..1]	A URI to the file containing this computing node generation image or file.
kind: GeneratorKind [0..1]	An optional attribute to define whether this generator uses a node image (i.e. a VM image) or a file (i.e. docker file) to generate the computing node.

Usage

The ComputingNodeGenerator is intended to be used as the common parent for all elements defining the characteristics of virtual computing nodes. Often these generators rely on a file which defines them.

4.11 VMImage Class

The VMImage class represents the image (i.e. the set of attributes and parameters) that can be used to generate a virtual machine.

Superclass

ComputingNodeGenerator

Associations

generatedVMs: VirtualMachine [0..*] The set of virtual machines that will be created using this image.

Usage

The VMImage is used for generation purposes, allowing the ICG to generate the scripts to generate VMs from a VM defining image.

4.12 ContainerImage Class

The ContainerImage class represents the image (i.e. the set of attributes and parameters) that can be used to generate a container.

Superclass

ComputingNodeGenerator

Attributes

generatedContainers: Container [0..*] The set of containers that have been generated using this container image.

Usage

The ContainerImage is used for generation purposes, allowing the ICG to generate the scripts to generate containers from a the container defining image.

4.13 AutoScalingGroup Class

The AutoScalingGroup class represents an aggregation of computing nodes with the auto scaling property.

Superclass

DOMLElement

Associations

supportedBy: RuntimeProvider [0..1] The runtime provider that supports the group.

groupedNodes: ComputingNode [0..*] The computing nodes

Usage

The AutoScalingGroup class allows to configure a set of nodes to act as a group.

4.14 Storage Class

The Storage class represents an infrastructure node that aims at incrementing the overall storage available to the computing nodes in the infrastructure.

Superclass

InfrastructureElement

Attributes

storage: Float [0..1]	The size of the storage in GB
cost: Float [0..1]	The cost of this storage service in Euro

Associations

ifaces: NetworkInterface [0..1]	The network interfaces connected to this infrastructure node.
---------------------------------	---

Usage

The Storage class allows to define a node that increments the storage of the application. The node cannot support any other functionality other than providing storage space.

4.15 FunctionAsAService Class

The FunctionAsAService class represents an pure software infrastructure component capable of executing functional algorithms through an API.

Superclass

InfrastructureElement

Attributes

cost: Float [0..1]	The cost of this service in Euro
--------------------	----------------------------------

Associations

ifaces: NetworkInterface [0..1]	The network interfaces connected to this infrastructure node.
---------------------------------	---

Usage

The FunctionAsAService class allows to define a service used to execute pure business logic/algorithms on a set of input data.

4.16 ExtInfrastructureElement Class (abstract)

The ExtInfrastructureElement class is just used to represent an instance of a new infrastructure element concept that the user wants to add to DOML. This class is part of the DOML-E extension mechanisms.

Superclass

InfrastructureElement, ExtensionElement

Usage

The ExtInfrastructureElement class is should be used to creat instances of concepts and metaclasses not currently available in DOML.

4.17 Network Class

The Network class represents the means to interconnect computing nodes. The concepts related to the network, as well as associations among them, is depicted in the following diagram:

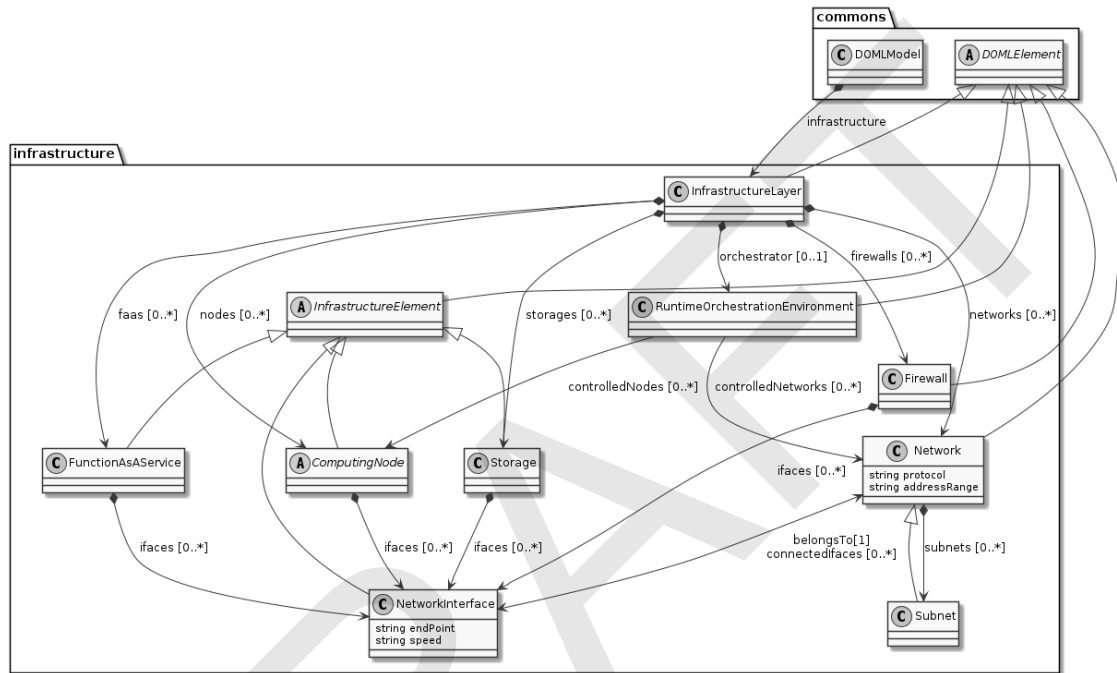


Figure 5. Infrastructure Layer diagram showing network related concepts

Superclass

DOMLElement

Attributes

protocol: String [0..1]	A string defining the protocol of the current network (e.g. TCP/IP).
addressRange: String [0..1]	A string describing the valid addresses in this particular network.

Associations

connectedifaces: NetworkInterface [0..*]	The set of network interfaces connected to this network. This is a derived association.
subnets: Subnet [0..*]	The set of sub networks of the current one.

Usage

The Network describes a means to interconnect computing nodes as part of a cloud architecture.

4.18 Subnet Class

The Subnet class models a partition of a main network. A subnet is also a network.

Superclass

Network

Usage

The Subnet is used to describe partitions of main networks.

4.19 NetworkInterface Class

The NetworkInterface class represents the means to interconnect computing nodes.

Superclass

InfrastructureElement

Attributes

endPoint: String [0..1]	A string defining the endpoint (i.e. address) of this network interface inside the network.
speed: String [0..1]	A string defining the maximum speed of this network interface.

Associations

belongsTo: Network [1]	A reference to the network associated to this interface.
------------------------	--

Usage

The Network describes a means to interconnect computing nodes as part of a cloud architecture.

4.20 Firewall Class

The Firewall class represents the means to interconnect computing nodes.

Superclass

DOMLElement

Associations

ifaces: NetworkInterface [0..*]	The interfaces of this Firewall element.
---------------------------------	--

Constraints

* Interfaces connected to a Firewall must be associated to different networks

Usage

The Firewall describes a device used to secure the access to a specific network.

4.21 RuntimeOrchestrationEnvironment Class

The RuntimeOrchestrationEnvironment class represents the environment that will be orchestrating the provisioning of computing nodes and the creation of the networks between these nodes.

Superclass

DOMLElement

Associations

controlledNetworks: Network [0..*] The networks controlled by the orchestrator.

controlledNodes: ComputingNode [0..*] The computing nodes controlled by the orchestrator.

Usage

The RuntimeOrchestrationEnvironment class may be used in the case PIACERE will support multiple orchestration environments. In this case, in fact, it will be possible to identify each environment in a DOML model and to allow users to select a specific one.

5 Concrete Layer

The following diagram shows the main elements of the Concrete Layer in DOML:

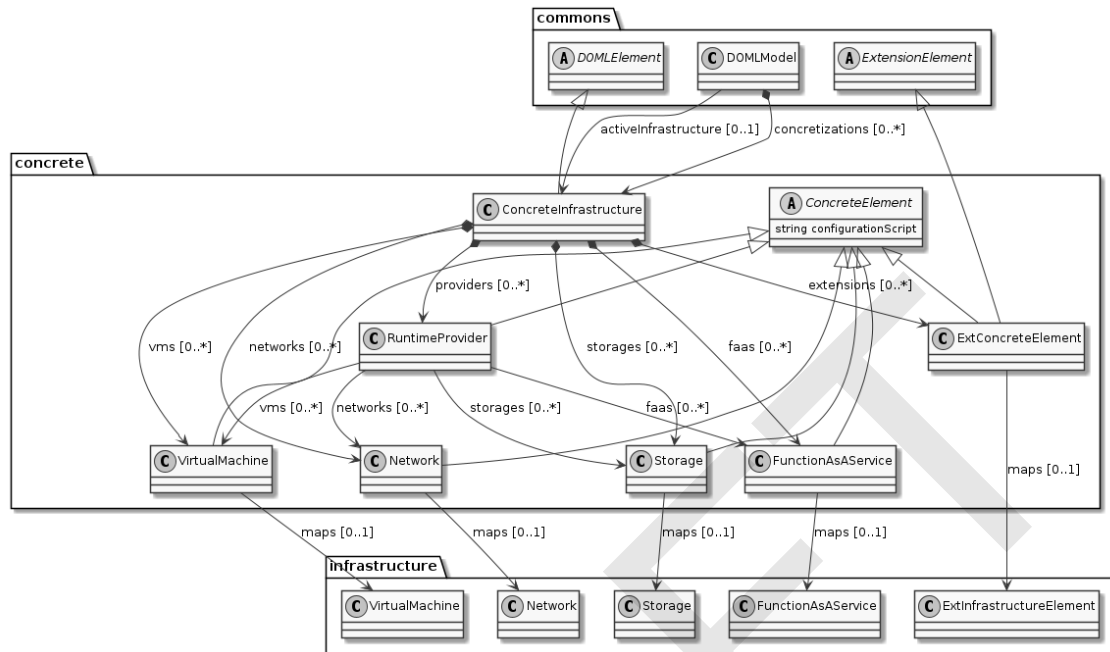


Figure 6. Infrastructure Layer diagram

5.1 ConcreteInfrastructure Class

The ConcreteInfrastructure class is the container for the catalog of concrete infrastructure elements that will be available to the current DOML configuration. Several concrete infrastructure instances may exist at the same time, each of them being part of a particular DOML solution.

Superclass

DOMLElement

Associations

providers:	RuntimeProvider	The list of runtime providers available in the catalogue [0..*]
nodes:	ComputingNode [0..*]	The list of independent computing nodes (not attached to a provider) available in the catalogue
generators:	ComputingNodeGenerator [0..*]	The list of virtual machine and container images available in the catalogue
asGroups:	AutoScalingGroup [0..*]	The list of independent auto scaling groups (not attached to a runtime provider) available in the catalogue
networks:	Network [0..*]	The list of independent networks (not attached to a runtime provider) available in the catalogue

storages: Storage [0..*]	The list of storage resources that will be part of this abstract infrastructure model
faas: FunctionAsAService [0..*]	The list of concrete function as a service nodes provided by this concrete infrastructure

Usage

The ConcreteInfrastructure is a container element, used as the catalog of the concrete infrastructure elements used to deploy the final cloud application using DOML for a particular solution.

5.2 ConcreteElement Class (abstract)

The ConcreteElement class represents all concrete infrastructure elements that can have an abstract infrastructure element component mapped onto them.

Superclass

DOMLElement

Attributes

configurationScript: [0..1]	String	An optional URI to the script that has to be executed to correctly configure a node.
-----------------------------	--------	--

Usage

The ConcreteElement is intended to be used as the parent for more concrete elements of the concrete infrastructure model.

5.3 RuntimeProvider Class

The RuntimeProvider class describes a cloud resources provider (e.g. AWS).

Superclass

DOMLElement

Associations

supportedGroups: AutoScalingGroup [0..*]	The groups requested to the runtime provider.
vms: VirtualMachine [0..*]	The virtual machines that will be provided by the runtime provider.
networks: Network [0..*]	The networks requested to the runtime provider.
storages: Storage [0..*]	The storages offered by this particular provided.
faas:FunctionAsAService [0..*]	The Faas services offered by this runtime provider

Usage

The RuntimeProvider is intended to model all the parameters related to a specific cloud IaaS provider.

5.4 VirtualMachine Class

The VirtualMachine class in the concrete layer represents a specific VM instance either provided by a runtime provider or configured by the user on their own infrastructure.

Superclass

ConcreteElement

Attributes

maps:	The VM on the abstract infrastructure layer this concrete
infrastructure.VirtualMachine	VM maps on.

Usage

The VirtualMachine is intended to be used as the concrete counterpart of the abstract VM defined in the infrastructure layer.

5.5 Network Class

The Network class in the concrete layer represents a specific network instance either provided by a runtime provider or configured by the user on their own infrastructure.

Superclass

ConcreteElement

Attributes

maps: infrastructure.Network	The network on the abstract infrastructure layer this concrete element maps on.
------------------------------	---

Usage

The Network is intended to be used as the concrete counterpart of the abstract Network defined in the infrastructure layer.

5.6 Storage Class

The Storage class in the concrete layer represents a specific storage service either provided by a runtime provider or configured by the user on their own infrastructure.

Superclass

ConcreteElement

Attributes

maps: infrastructure.Storage The storage service on the abstract infrastructure layer this concrete storage maps on.

Usage

The Storage is intended to be used as the concrete counterpart of the abstract Storage defined in the infrastructure layer.

5.7 FunctionAsAService Class

The FunctionAsAService class in the concrete layer represents a specific functional logic service instance either provided by a runtime provider or configured by the user on their own infrastructure.

Superclass

ConcreteElement

Attributes

maps: infrastructure. The faas instance on the abstract infrastructure layer this FunctionAsAService concrete element maps on.

Usage

The FunctionAsAService is intended to be used as the concrete counterpart of the abstract FunctionAsAService defined in the infrastructure layer.

5.8 AutoScalingGroup Class

The AutoScalingGroup class in the concrete layer represents a specific group instance either provided by a runtime provider or configured by the user on their own infrastructure.

Superclass

ConcreteElement

Attributes

maps: infrastructure.AutoScalingGroup The group on the abstract infrastructure layer this concrete group maps on.

Usage

The AutoScalingGroup is intended to be used as the concrete counterpart of the abstract AutoScalingGroup defined in the infrastructure layer.

5.9 ExtConcreteElement Class

The ExtConcreteElement class is just used to represent an instance of a new infrastructure element concept that the user wants to add to DOML. This class is part of the DOML-E extension mechanisms.

Superclass

ConcreteElement, ExtensionElement

Usage

The ExtConcreteElement class is should be used to creat instances of concepts and metaclasses not currently available in DOML.

DRAFT

6 Optimization Layer

The following diagram shows the main elements of the Optimization Layer in DOML:

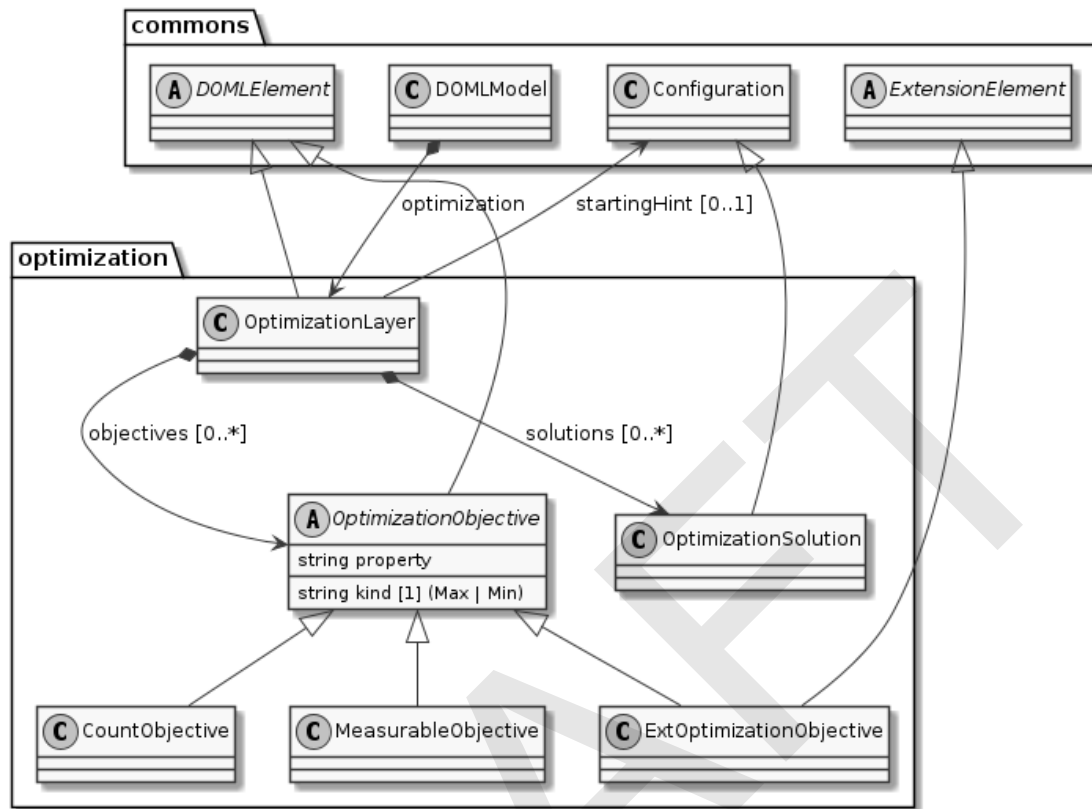


Figure 7. Optimization Layer diagram

6.1 OptimizationLayer Class

The OptimizationLayer class is the main container for all the elements related to the definition and usage of the optimization algorithms in DOML.

Superclass

DOMLModel

Associations

objectives:	OptimizationObjective	The set of objectives for the optimization algorithms.
	[0..*]	
solutions:	OptimizationSolution	All the solutions generated by the optimization algorithm.
	[0..*]	
startingHint:	Configuration	An optional configuration instance that will be used as a hint by the optimization algorithm.
	[0..1]	

Constraints

* At least one optimization objective should be provided to be able to use the model for optimization purposes.

Usage

The OptimizationLayer is intended to be used as a container for the objectives and solutions associated to the optimization algorithms for a DOML model.

6.2 OptimizationObjective Class (abstract)

The OptimizationObjective class represents a formal objective for an optimization algorithm. This objective will afterwards be used by the algorithms as an input to obtain a solution for the application deployment into the cloud infrastructure.

Superclass

DOMLElement

Attributes

property: String [1]	The property associated to this optimization objective.
kind: String [1]	The kind of objective, which can be either “max” or “min”.

Constraints

* The kind attribute may only have the “min” or “max” values.

Usage

The OptimizationObjective is made abstract to serve as the basis for more concrete optimization objectives, such as objectives that measure a property, or objectives that are related to counting the number of different values of a property.

6.3 CountObjective Class

The CountObjective class represents an optimization objective that will count the different number of values associated to the property specified on them.

Superclass

OptimizationObjective

Usage

The CountObjective is used to define optimization objectives which want to maximize or minimize the total number of values a property may take (e.g. minimize the number of locations for all the servers in a DOML solution).

6.4 MeasurableObjective Class

The MeasurableObjective class represents an optimization objective associated to the measurement of a particular property.

Superclass

OptimizationObjective

Usage

The MeasurableObjective is used to define an optimization objective related directly to the value of a particular property (e.g. minimize the cost or maximize the throughput of a DOML solution).

6.5 OptimizationSolution Class

The OptimizationSolution class represents a Configuration of the current DOML model obtained through the usage of optimization algorithms.

Superclass

Configuration

Usage

The OptimizationSolution is a subclass of the main Configuration class in the commons package, as it is foreseen that any information related to the results obtained by the optimization algorithms (e.g. parameters, used requirements, etc.) could be added as additional information to this kind of Configuration instances.

6.6 ExtOptimizationObjective Class (abstract)

The ExtOptimizationObjective class is just used to represent an instance of a new optimization objective concept that the user wants to add to DOML. This class is part of the DOML-E extension mechanisms.

Superclass

OptimizationObjective, ExtensionElement

Usage

The ExtOptimizationObjective class should be used to create instances of concepts and metaclasses not currently available in DOML.

7 DOML Text Syntax

The following figures define the current DOML syntax. This will evolve in the future releases based on the feedback by end users and the other PIACERE technical partners.

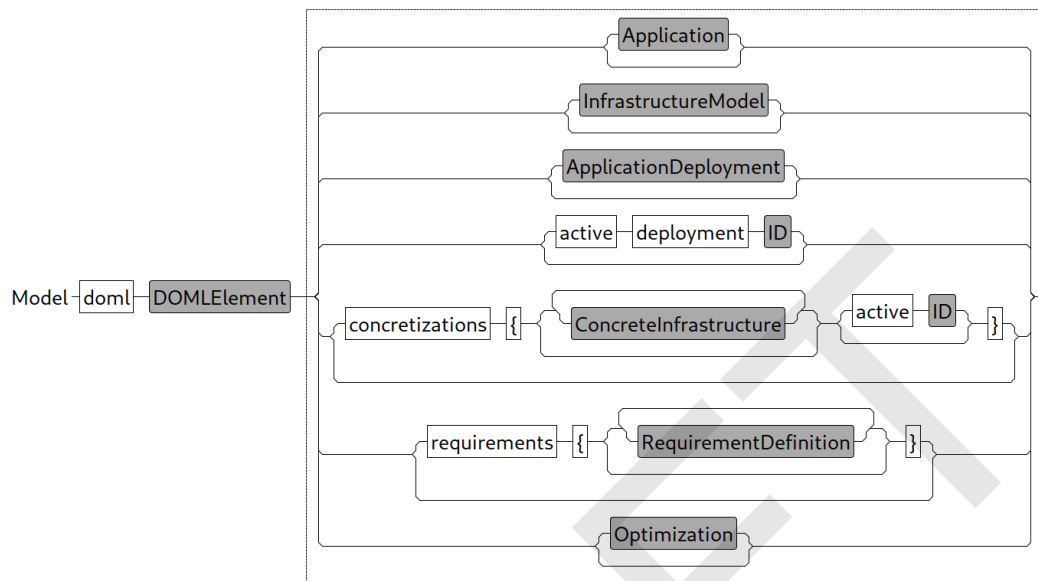


Figure 1. DOML top level model.

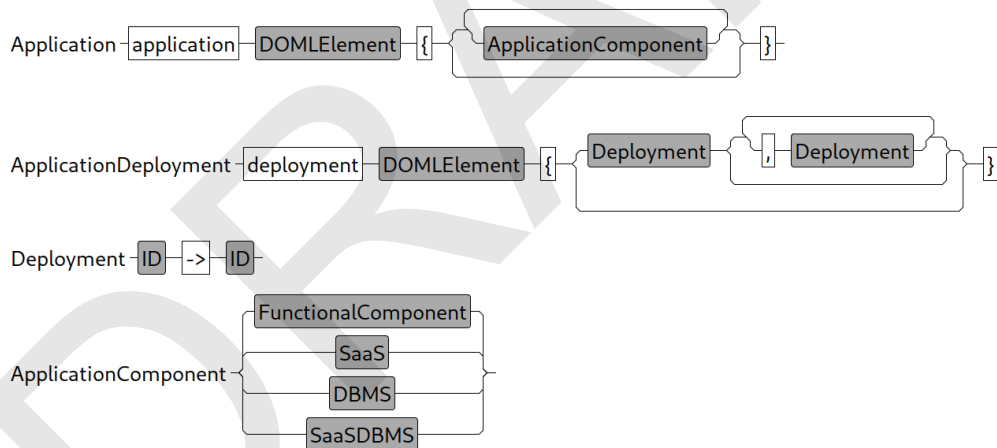


Figure 2. Application layer model.

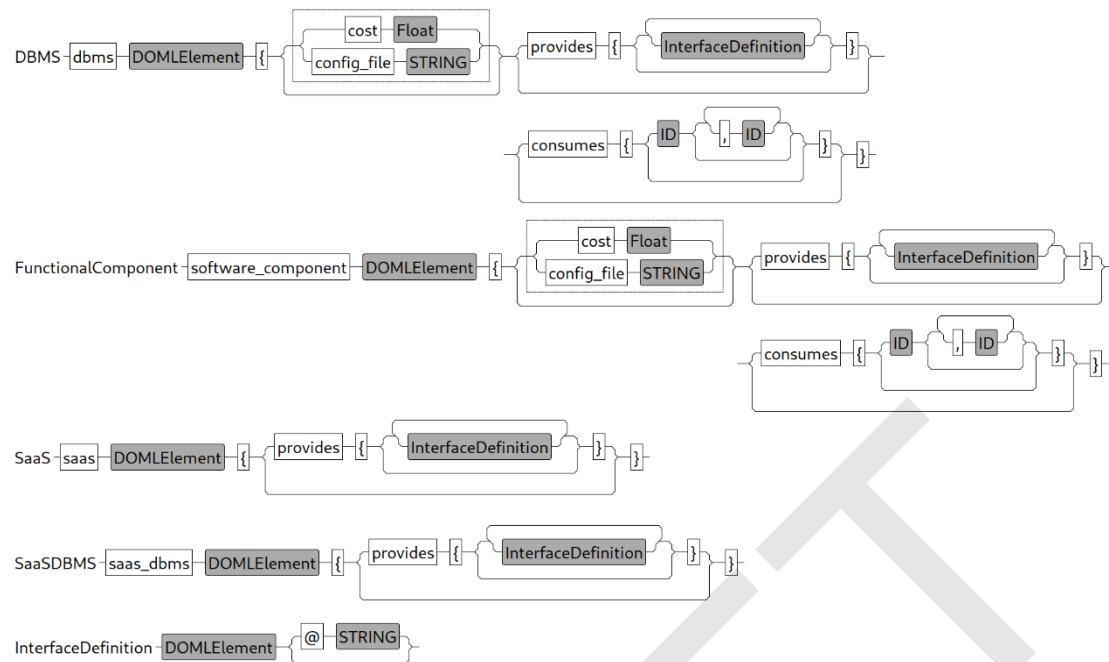


Figure 3. Application Components model.

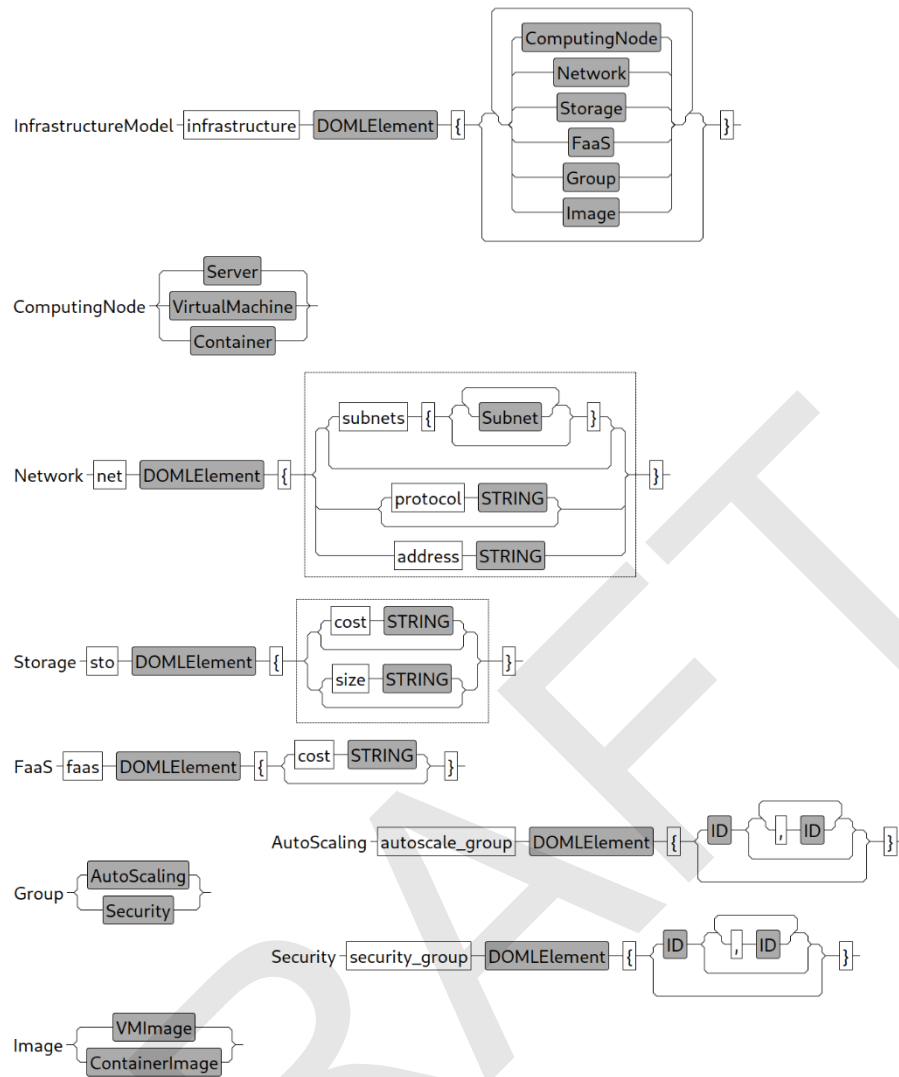


Figure 4. Abstract Infrastructure layer model.

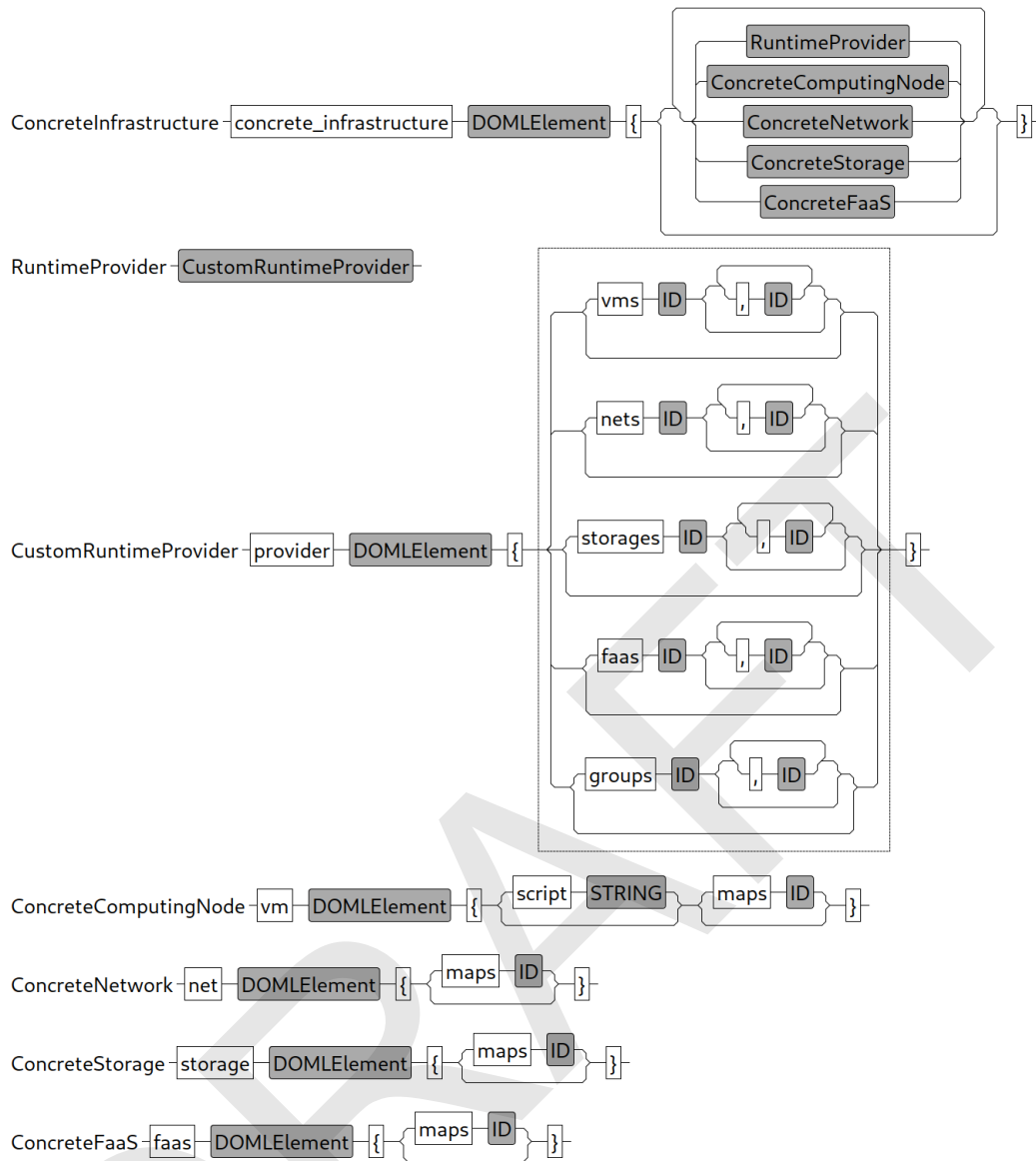


Figure 5. Concrete Infrastructure layer model.

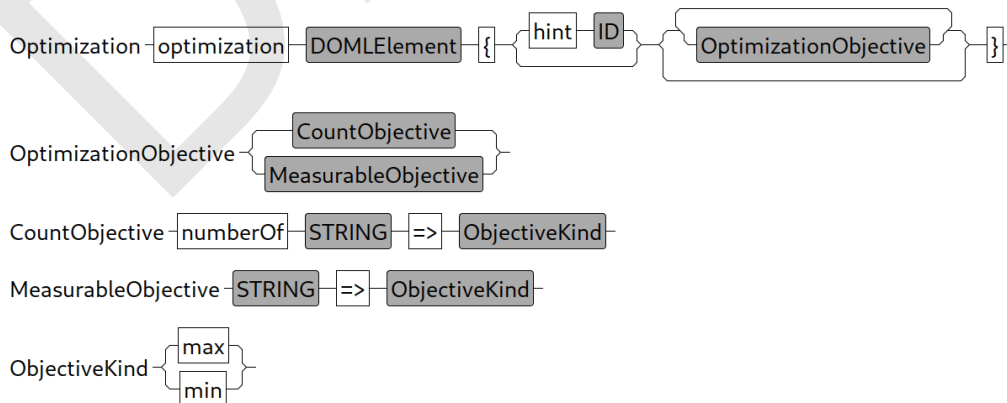


Figure 6. Optimization layer model.

8 DOML Examples

In this section we provide two simple DOML examples, showing their definition in the textual syntax and the corresponding translation in an XMI notation. Other examples are available in Deliverable D3.1.

8.1 Simple Web Application

This example describes a very simple DOML model with a web application that accesses an external API and a database and 2 IoT nodes that provide information to the application. The main software components will be running on virtual machines provided by a runtime provider, while the *iotUnits* will be running on physical nodes. The configuration has been made manually by the user.

A possible textual representation of this DOML model would be as follows:

```
doml simple
application simpleApp {
  dbms oracle {
    provides {
      db
    }
  }
  functional webapp {
    provides {
      logMessage
    }
    consumes { db, getWeather }
  }
  functional iotProvider {
    consumes { logMessage }
  }
  saas meteoAPI {
    provides {
      getWeather @ "https://api.mymeteo.com/get"
    }
  }
}
infrastructure infra {
  provider AWS {
    vms vm1, vm2
  }
  vm vm1;
  vm vm2;
  node iotNode1;
  node iotNode2;
}
deployment config1 {
  oracle -> vm1,
  webapp -> vm2,
  iotProvider -> iotNode1,
  iotProvider -> iotNode2
}
active config1
```

The example above shows the 2 main layers of DOML: application and infrastructure, as well as some elements in the commons layer (i.e. the configuration). As described in the example, the application layer contains 4 application components:

- The Oracle database software, defined by the DBMS metaclass. This component provides a software interface to access the database.
- The main web application software component, defined by the SoftwarePackage metaclass. As described in the example definition, the component consumes the

database interface and the external meteoAPI service. It also provides an interface for IoT components to send messages.

- The IoT software component, which will be deployed to all IoT nodes and that uses the logMessage interface of the web application to upload their data.
- The external API required by the web application, described by the SaaS metaclass. In the case of the external API, the service that it provides has been given an end point using a URL.

The infrastructure layer, which defines all the infrastructure elements catalogue available for the application layer, is also modelled as describen in the example:

- Two virtual machines are defined.
- The AWS runtime provider is also described, as the provider for the virtual machines.
- Two physical nodes are modelled as the IoT nodes deployed with the application.

Finally, a configuration is done by simply linking the application component to the corresponding infrastructure elements, and that configuration is configured as the active configuration.

The XML representation of the above model is shown below:

```
<?xml version="1.0" encoding="ASCII"?>
<commons:DOMLModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:app="http://www.piacere-
project.eu/doml/application" xmlns:commons="http://www.piacere-project.eu/doml/commons"
xmlns:infra="http://www.piacere-project.eu/doml/infrastructure" name="simple"
activeConfiguration="//@configurations.0">
  <application name="simpleApp">
    <components xsi:type="app:DBMS" name="oracle">
      <exposedInterfaces name="db"/>
    </components>
    <components xsi:type="app:SoftwarePackage" name="webapp"
consumedInterfaces="//@application/@components.0/@exposedInterfaces.0
//@application/@components.3/@exposedInterfaces.0">
      <exposedInterfaces name="logMessage"/>
    </components>
    <components xsi:type="app:SoftwarePackage" name="iotProvider"
consumedInterfaces="//@application/@components.1/@exposedInterfaces.0"/>
    <components xsi:type="app:SaaS" name="meteoAPI">
      <exposedInterfaces name="getWeather" endPoint="https://api.mymeteo.com/get"/>
    </components>
  </application>
  <infrastructure name="infra">
    <providers name="AWS" providedVMs="//@infrastructure/@nodes.0
//@infrastructure/@nodes.1"/>
    <nodes xsi:type="infra:VirtualMachine" name="vm1"/>
    <nodes xsi:type="infra:VirtualMachine" name="vm2"/>
    <nodes xsi:type="infra:PhysicalComputingNode" name="iotNode1"/>
    <nodes xsi:type="infra:PhysicalComputingNode" name="iotNode2"/>
  </infrastructure>
  <configurations name="config1">
    <deployments component="//@application/@components.0"
node="//@infrastructure/@nodes.0"/>
    <deployments component="//@application/@components.1"
node="//@infrastructure/@nodes.1"/>
    <deployments component="//@application/@components.2"
node="//@infrastructure/@nodes.2"/>
    <deployments component="//@application/@components.2"
node="//@infrastructure/@nodes.3"/>
  </configurations>
</commons:DOMLModel>
```

8.2 Optimization Problem Example

This example describes a DOML model that will be fed to an optimization service to obtain an optimal configuration for the cloud application according to a set of formal requirements. In this case the software components need to be deployed into some/all of the infrastructure elements in the catalogue.

The textual representation of the DOML model is shown below:

```
doml ^optimization
application optimizationApp {
    functional comp1 {

    }
    functional comp2 {

    }
    functional comp3 {

    }
}
infrastructure catalogue {
    provider AWS {
        vms size1, size2
    }
    vm size1 "$cost"="50", "$performance"="100", "$location"="Europe";
    vm size2 "$cost"="100", "$performance"="200", "$location"="America";
    provider Google {
        vms g1, g2
    }
    vm g1 "$cost"="15", "$performance"="50", "$location"="Europe";
    vm g2 "$cost"="75", "$performance"="120", "$location"="Asia";
}
optimization opt {
    numberOf "location" => min
    "performance" => max
    "cost" => min
}
```

As described before, the DOML captures all 3 software components in the application, and proposes 4 different virtual machines, from 2 different providers, that will be used to deploy the application. The latter model should be then sent as an input to the optimization algorithms to obtain a configuration that matches the objectives described in the optimization layer.

The equivalent XML definition of the DOML model is shown below:

```
<?xml version="1.0" encoding="ASCII"?>
<commons:DOMLModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:app="http://www.piacere-
project.eu/doml/application" xmlns:commons="http://www.piacere-project.eu/doml/commons"
xmlns:infra="http://www.piacere-project.eu/doml/infrastructure"
xmlns:optimization="http://www.piacere-project.eu/doml/optimization"
name="optimization">
  <application name="optimizationApp">
    <components xsi:type="app:SoftwarePackage" name="comp1"/>
    <components xsi:type="app:SoftwarePackage" name="comp2"/>
    <components xsi:type="app:SoftwarePackage" name="comp3"/>
  </application>
  <infrastructure name="catalogue">
    <providers name="AWS" providedVMs="//@infrastructure/@nodes.0
//@infrastructure/@nodes.1"/>
    <providers name="Google" providedVMs="//@infrastructure/@nodes.2
//@infrastructure/@nodes.3"/>
    <nodes xsi:type="infra:VirtualMachine" name="size1">
      <annotations key="cost" value="50"/>
      <annotations key="performance" value="100"/>
      <annotations key="location" value="Europe"/>
    </nodes>
    <nodes xsi:type="infra:VirtualMachine" name="size2">
```

```
<annotations key="cost" value="100"/>
<annotations key="performance" value="200"/>
<annotations key="location" value="America"/>
</nodes>
<nodes xsi:type="infra:VirtualMachine" name="g1">
  <annotations key="cost" value="15"/>
  <annotations key="performance" value="50"/>
  <annotations key="location" value="Europe"/>
</nodes>
<nodes xsi:type="infra:VirtualMachine" name="g2">
  <annotations key="cost" value="75"/>
  <annotations key="performance" value="120"/>
  <annotations key="location" value="Asia"/>
</nodes>
</infrastructure>
<optimization name="opt">
  <objectives xsi:type="optimization:CountObjective" kind="min" property="location"/>
  <objectives xsi:type="optimization:MeasurableObjective" kind="max"
property="performance"/>
  <objectives xsi:type="optimization:MeasurableObjective" kind="min" property="cost"/>
</optimization>
</commons:DOMLModel>
```


9 Conclusions

This document has described the specification of the DOML language. The DOML has been conceived as a declarative language to make it easier for non-expert users, but it includes mechanisms to include imperative scripts and advanced features for expert user profiles.

DOML has also been designed taking into account the fast evolution of the cloud computing state-of-the-art, including mechanisms to extend itself easily, adding more concepts and properties to existing ones.

DRAFT