ARTIST

FP7-317859

Advanced software-based seRvice provisioning and migraTIon of legacy Software

---

Deliverable D9.6

Automated Deployment Strategies

Editor(s):	Jesús Gorroñogoitia
Responsible Partner:	ATOS
Status-Version:	v1.0
Date:	31/03/2015
Distribution level (CO, PU):	PU

Project Number:	FP7-317859
Project Title:	ARTIST

Title of Deliverable:	Automated Deployment Strategies
Due Date of Delivery to the EC:	31/03/2015

Workpackage responsible for the Deliverable:	WP9 - New software generation by forward engineering
Editor(s):	ATOS
Contributor(s):	TUWIEN, SPIKES, ICCS, INRIA
Reviewer(s):	Kleopatra Konstanteli (ICCS)
Approved by:	All Partners
Recommended/mandatory readers:	WP6, WP7, WP9, WP12

Abstract:	This deliverable comprises automatically executable transformations needed for deploying the modernized applications in specific Cloud infrastructures
Keyword List:	Forward Engineering, Model Transformation, Deployment
Licensing information:	Generally EPL (open source), indicated otherwise. The document itself is delivered as a description for the European Commission about the released software, so it is not public.

Document Description

Document Revision History

Version	*Date*	*Modifications Introduced*	
		Modification Reason	*Modified by*
v0.1	24/02/15	ToC, First draft version	ATOS, TUWIEN, ICCS, SPIKES, INRIA
v0.2	10/03/15	Contributions	TUWIEN, ATOS
V0.3	11/03/15	Contributions	TUWIEN, ICCS, ATOS
V0.4	12/03/15	Contributions	TUWIEN, ATOS
V0.5	13/03/15	Peer-Review Version	ATOS
V0.6	25/03/15	Peer-Review	ICCS
V1.0	31/03/15	Final version	ATOS, ICCS, TUWIEN

Table of Contents

Table of Figures

Table of Tables

NO TABLE OF FIGURES ENTRIES FOUND.

Terms and abbreviations

ATL	Atlas Transformation Language
AWS	Amazon Web Services
CAML	Cloud Application Modelling Language
CCUI	Command and Console User Interface
DBMS	Database Management System
DEWS	Distance Early Warning System
DSL	Domain Specific Language
EAR	Enterprise Archive
EE	Enterprise Edition
EMF	Eclipse Modelling Framework
EPL	Eclipse Public License
GAE	Google App Engine
GUI	Graphical User Interface
IaaS	Infrastructure as a Service
JAR	Java Archive
LoB	Line of Business
M2M	Model to Model
M2MT	Model to Model Transformation
M2T	Model to Text
MDTB	Model Discovery Toolbox
MUTB	Model Understanding Toolbox
OVF	Open Virtualization Format
PaaS	Platform as a Service
PDM	Platform Description Model

PI	Platform Independent
PSM	Platform Specific Language
SDK	Software Development Kit
UI	User Interface
UML	Unified Modelling Language
VM	Virtual Machine
WAR	Web Archive
WP	Work Package
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Executive Summary

Task 9.5 'Deployment patterns expressed as transformations' manages one of the activities required during the modernization phase of the ARTIST Methodology, performed on the components of an application being migrated to the Cloud. Once the models (PSM) that described the application components have been "cloudified" and optimized (resulting on modernized models) and the corresponding target source code has been generated (out of these modernized models), there is still an important task remaining, since the modernized components need to be deployed into the target Cloud environment. This deployment process is conducted and personalized by a set of descriptor files, which require to be generated for the selected target Cloud environment. These descriptors customize the deployment process by expressing concrete deployment needs on the target and configuring the required Cloud services. Moreover, the modernized components can only be deployed, depending on the target, when packaged into concrete deployment units, which are typically compliant with strict deployment standards, which are Cloud or application dependent (like for instance WAR or EAR files for J2EE platform deployment or OVF for VM packaging).

The deployment process itself is supported by the target Cloud environment, normally through a target-dependent SDK, which offers UI tools that enable the deployment process. Most of the deployment SDKs offer command-line scripts that enable the manual deployment process (intended for advanced users), although many Cloud providers offer as well GUIs (either standalone ones or integrated within popular IDEs or Web-accessible ones) that simplify the deployment process for not-advanced users.

As a consequence, the task to deploy a modernized application (or certain components) requires (for most of the Cloud environments, either infrastructure or platform) the elaboration of a set of artefacts supporting the deployment process itself.

Before supporting this process, application owners need to identify the Cloud target provider and its offerings, based on the selection of Cloud services, required by the application. In this context, the Cloud Target Selection Tool, specified in this document, supports the application owner in the identification of required Cloud services and their matching against the available Cloud provides, described by the available CloudML@ARTIST models.

Moreover, this model-driven deployment approach relies on the availability of deployment models for the application. These models include generic deployment specifications, based on UML complemented with the platform-independent CloudML@ARTIST meta-model, and Cloud provider specifications, (included in CloudML@ARTIST as well).

The deployment tool enables the semi-assisted generation of deployment artefacts, out of these models, assuming that they already contain deployment information and requirements. In a similar approach followed by T9.4 Target Generation, this tool produces these textual-based deployment artefacts by obtaining the deployment requirements and configuration from the models.

ARTIST focuses on the model-driven migration support for selected target Cloud providers, namely Google App Engine (GAE), and Microsoft Azure. Nonetheless, the ARTIST model-driven deployment support will be investigated towards achieving the most generic and extensible deployment support possible, compatible with the specific characteristics of these Cloud providers.

This document reports the specification, design and implementation of the Cloud Target Selection and Deployment Tools, and also elaborates on the model-based specification of the deployment. Additionally, it reports on the preliminary research analysis that was conducted

for the design and implementation of the tools, as well as the conceptual deployment strategies identified during this research.

Modelling deployment in UML, using CloudML@ARTIST is explained by example, using the ARTIST Petstore use case. Deployment for the other ARTIST use cases (i.e. DEWS and Spikes LoB) is modelled as well, as described in the user manual section of the Deployment Tool. Reusable deployment templates for concrete Cloud providers and the interoperability of the ARTIST deployment modelling approach with others are further elaborated.

Cloud Target Selection Tool and Deployment Tool specifications (both functional and technical) are provided, including components and implementation details. For both tools, a walk-through user guide is provided. For the Deployment Tool, this user guide describes its usage to generate deployment descriptors for the Google App Engine and the Microsoft Azure Cloud platforms, for DEWS and LoB use cases, respectively.

The implementation of the Deployment Tool provides specific support for deployment on Google App Engine and Microsoft Azure, the default target Cloud platforms of choice for ARTIST use cases. No specific implementation support for Amazon Web Services (AWS) was implemented, despite it was included in the preliminary analysis, mainly for the following two reasons:

- AWS support has not being required by ARTIST use cases for deployment
- Deployment in AWS is already supported by the Cloud platform deployment framework of ModaCloud, based on CloudML. ARTIST provides a translator from CloudML@ARTIST to CloudML@ModaCloud, whereby deployment models for AWS created within ARTIST can be translated and used to support AWS deployment using the ModaCloud framework.

Nonetheless, the ARTIST deployment approach is founded on solid and generic enough model-driven techniques, underpinning a common deployment modelling language, which enables the adaptation, with little effort on implementation, of the Deployment Tool to support other Cloud target environments.

1 Introduction

1.1 About this deliverable

The modelling of the application deployment to the Cloud takes place during modernisation phase of the ARTIST Methodology. It happens once the models that described the application components have been modernised and optimized, and their source code, compatible with the target Cloud offering, has been generated (from these modernised models). Then, there is still an important activity remaining, since the application components need to be deployed into the target Cloud environment.

The deployment activity is supported and configured by a set of deployment descriptors. These descriptors personalized the deployment process by expressing concrete deployment configurations for the target Cloud environment and by configuring the required services. Additionally, the modernized components are deployed, depending on the target, bundled within specific deployment units, which are compliant with formal deployment standards, some of them Cloud specific (OVF for VM packaging), but other application specific (EAR for J2EE platform).

The deployment process is typically Cloud provider specific, driven by vendor specific procedures and mechanisms, although supported by SDKs, which offer tools that simplify the deployment process, including GUIs, either standalone or integrated with popular IDEs or Web-accessible. Most of these deployment tools also offer command-line scripts, targeting advance users, enabling a manual deployment process.

In the particular case of Cloud infrastructure providers, there are public common libraries that offer a common SDK supporting deployment on a wide range of providers through the same interface, such as Apache jClouds[1] or libCloud[2] .Up to our knowledge, there is no similar initiative concerning Cloud platforms, although some attempts to standardize a common Cloud platform interface have already been started (i.e. CAMP[3]).
As a consequence, the task to deploy a modernized application (or certain components) requires (for most of the Cloud environments, either infrastructure or platform) the elaboration of a set of artefacts supporting the deployment process itself:

- One or more deployment descriptors, which are typically Cloud provider specific.
- One or more deployment units (i.e. bundles), which are either application- (e.g. framework) or Cloud provider-specific.
- One or more deployment scripts (i.e. invocations of a deployment SDK).

Before supporting the deployment process itself, application owners requires identifying the Cloud target provider and its offering, based on the selection of Cloud services required by the application. In this context, the Cloud Target Selection Tool supports the application owner in the identification of required Cloud services and the matching of these requirements against the available Cloud provides, identified by the CloudML@ARTIST models available.
Moreover, the Model-Driven deployment approach, described in this document, relies on the availability of deployment models for the application. These models include generic deployment specifications, based on UML complemented with CloudML@ARTIST, and specifications Cloud provider specifications, based on CloudML@ARTIST.

[1] http://jclouds.apache.org/
[2] http://libcloud.apache.org/
[3] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=camp

T9.5 'Deployment patterns expressed as transformations' enables the semi-assisted generation of these deployment artefacts, as introduced in the previous enumeration, out of these deployment models. In a similar approach followed by T9.4 'Target Generation', this task produces these textual-based deployment artefacts, by inspecting the deployment requirements and configuration for the modernized application.

However, T9.5 does NOT support the deployment process itself: this task does not provide a homogeneous and universal deployment SDK supporting a wide range of target Cloud environments. Nevertheless, T9.5 can provide (for some selected target Cloud environments offering a command-line SDK) some tailored scripts which, once executed by end-users, deploy the modernized application (or concrete components) into the target Cloud environment. It is implied that the end-user has previously installed an operative target Cloud deployment framework.

In a wider context, the deployment of complex distributed applications on the Cloud has been increasingly receiving attention in the last years [1]. ARTIST focuses on the model-driven migration support for selected target Cloud providers, namely Google App Engine (GAE) and Microsoft Azure. Nonetheless, the ARTIST model-driven deployment support will be investigated towards achieving the most generic and extensible deployment support possible, compatible with the specific characteristics of these Cloud providers.

This document reports the specification, design and implementation of the Cloud Target Selection Tool and the Deployment Tool. It also depicts the CloudML@ARTIST approach to model deployment requirements. Additionally, it reports on the preliminary research analysis that was conducted for the design and implementation of these tools, as well as the conceptual deployment strategies identified during this research.

The implementation of the Deployment Tool provides specific support for deployment on Google App Engine and Microsoft Azure, the default target Cloud platforms of choice for ARTIST use cases. No specific implementation support for Amazon Web Services (AWS) was implemented, despite although it was included in the preliminary analysis, mainly for the following two reasons:

- AWS support seems not being required by ARTIST use cases for deployment
- Deployment in AWS is already supported by the Cloud platform deployment framework of ModaCloud [27], based on CloudML. ARTIST provides a translator from CloudML@ARTIST to CloudML@ModaCloud [28], whereby deployment models for AWS created within ARTIST can be translated and use to support AWS deployment using the ModaCloud framework.

1.2 Document structure

This document is structured as follows. Section 2 provides the functional and technical specification of the Cloud Target Selection Tool, which enables users to determine the most suitable target Cloud offering where to deploy the migrated application, based on browsing the services offered by the Cloud provider. Section 3 describes the operational approach for deployment modelling based on CloudML@ARTIST and UML. Section 4 provides the functional and technical specification of the Deployment Tool, which enables the users to create required deployment descriptors and scripts from deployment models, for selected target Cloud offerings. Section 5 provides details for the delivery and usage of the Cloud Target Selection

and Deployment tools, as well as their user manuals. Section 6 concludes the document outlining the main results of this work, and foreseen future work and improvements. APPENDIX A reports the analysis of survey conducted on the state of the art on model-driven support for deployment to Cloud. APPENDIX B reports on the analysis of the deployment strategies for the reference Cloud offerings: Google App Engine, Amazon Web Services and Microsoft Azure. APPENDIX C reports the research conducted aiming at detecting platform independent deployment patterns (from the analysed ones) as well as designing a platform independent meta-model for expressing conceptually these deployment patterns. Based on this meta-model, platform description models (PDMs) for reference Cloud offerings were derived.

1.3 Fitting into the overall ARTIST solution

In the following, we describe from a functional point of view how the Deployment Tool fits into the overall ARTIST migration tool suite. The Deployment Tool is developed in the context of WP9 and considered as part of the forward engineering process. Models reverse-engineered by the MDTB and the MUTB – both are provided by WP8 – and modernized by the MCF are considered as input of modelling the deployment. In fact, the component viewpoint is of primarily interest for producing deployment models. Components are materialized by artefacts that are deployed onto a Cloud environment. Hence, the quality of the reverse-engineered and modernized components plays a crucial role. Only if they are appropriately designed, the quality of the deployment models will be sufficient and useful to generate deployment scripts required for initiating the resource provisioning at the target cloud environment. While producing deployment models is considered to be mainly a manual task, the generation of deployment scripts for a certain cloud environment is supported by dedicated model transformations. To select an appropriate cloud provider and guide the developer through this complex decision making process, the Cloud Target Selection tool is provided. It builds on information captured by CloudML@ARTIST developed in WP7.

Figure 1 ARTIST Overall Deployment process

Figure 1 outlines the overall ARTIST deployment process. By using the Cloud Target Selection tool (cf. Section 2), users can determine the Cloud providers that better fits their requirements expressed by selecting the Cloud services they are looking for. Once the Cloud target has been selecting, they can use CloudML@ARTIST to model the concrete deployment requirements and topology for their application components (cf. Section 3). Finally, they can use the deployment tool to generate target-specific deployment descriptors from the former models (cf. Section 4).

1.4 Main Innovations

The main focus of this deliverable is to report on the languages, techniques, and components applied to produce cloud-based deployment models, select suitable Cloud providers that offers required services and generate the deployment descriptors required to deliver to Cloud the migrated application. Main innovations in this respect can be summarized as follows.

Cloud Target Selection. The main innovations of Cloud Target Selection tool lie in the challenge of exploiting the CloudML@ARTIST model structure and can be concluded as follows:

- It contains a fully extensible mechanism which supports the filtering of the information contained within the meta-models and providing it to the end user in a human-consumable format through a user-friendly environment. From the extensibility point of view, extensions can be easily made in order to support even more perspectives of the existing underlying models, or extensions of them. E.g. legal aspects, and benchmark results.
- The information exchange taking place through the interaction with the user, does not require but a minimum set of knowledge about the elements described and defined in the meta-models. This means that, in the future, potential changes in the cloud providers' descriptions or even in the definitions of general cloud environments in the context of CloudML@ARTIST meta-model, will not result in non-compatibility with the tool. On the contrary, new features can automatically be detected and take their place in the equation solving the problem of selecting cloud target.
- It allows the user to express their requirements against the target cloud environments, thus fully utilizing and exploiting all the benefits coming from the construction of CloudML@ARTIST modelling language.
- Users' expression of cloud requirements immediately and automatically result in reports and suggestions about the best matching supported provider. Otherwise, this task, even with the use of the same meta-models, would take a lot of time and effort while it would absolutely require a good level of modelling knowledge in general and in particular about the specific meta-models under use.

Modelling Cloud-based Deployments in UML. Current cloud modelling approaches [2] address the diversity of cloud environments by introducing a considerable set of modelling concepts in terms of novel domain-specific languages. At the same time, general-purpose languages, such as UML, provide modelling concepts to represent software, platform and infrastructure artefacts from different viewpoints where the deployment view is of particular relevance for specifying the distribution of application components on the targeted cloud environments. The generic nature of UML's deployment language calls for a cloud-specific extension to capture the plethora of cloud services at the modelling level. For that reason we developed the Cloud Application Modelling Language (CAML) as part of CloudML@ARTIST to enable cloud-based deployments to be specified directly in UML. This is especially beneficial for migration scenarios where reverse-engineered UML models are tailored towards a selected cloud environment as it is advocated by ARTIST.

Deployment Tools for the Cloud. As it is described in this document (see APPENDIX A and B), the existence of a plethora of Cloud providers, with their offerings and services, imposes a diversity of deployment approaches, interfaces and services. Initiatives to harmonize this diversity, offering a common deployment interface have focused mainly on supporting the

Cloud infrastructure, but only recently standardization initiatives to harmonize the deployment on the platform have started. In this work, we show how a common modelling language supporting deployment specifications (UML combined with CloudML@ARTIST) can be exploited to generate the required deployment descriptors for a selection of Cloud providers (i.e. Google App Engine, Microsoft Azure), by developing model-to-model transformations, which convert deployment models, compliant to UML-CloudML@ARTIST into those compliant to the specific Cloud meta-models describing those descriptors. A set of model-to-text serializers, also developed in this work, generate the descriptors. This approach is generic, and can be extended to support the deployment into any other Cloud provider, with low effort.

2 Cloud Target Selection Tool

The Cloud Target Selection tool is an Eclipse plugin aiming at guiding the user through the complex process of making the final decision for the target platform of the migration of their application or application components. This tool targets developers who have a good insight of the application's technical and functional needs, but little knowledge about the services or the benefits coming from the cloud platform providers. This case is quite common for developers with little experience on cloud, or with much experience on a single cloud vendor's platform, as the tasks of browsing through the countless documentation sheets and web-pages, as well as of modelling and combining the resulting information, in order to get to the final decision, are confusing and time consuming.

2.1 Functional description

The Cloud Target Selection Tool aims at exploiting the information lying in the meta-models, in order to make a suggestion about the target platform that best fits the needs of the application to be migrated. More precisely, the functionality of the tool, as a whole, can be summarised in three main activities:

- It supports the visualization of the main features and service characteristics of the target platforms, as they have been identified and modelled in the context of ARTIST project.
- It allows the selection of the ones - of the above features - that are considered to be important offerings for the specific application's needs, through a user-driven process. This way, the user can actually set the target platform standards for hosting the components of the application.
- It performs a matching process between the user's selection and the actual real-world target platform offerings, so as to indicate the best fitting solution among the available choices.

As described in Figure 2, the actions to be performed from the user’s perspective are:

- **Action 1:** Open the Eclipse views showing features and services.
- **Action 2:** Select the desired ones from the tree-structures of these views.
- **Action 3:** Indicate the candidate cloud providers among the supported ones by the CloudML@ARTIST.
- **Action 4:** Ask for a validation to be performed over the selections made in previous steps.

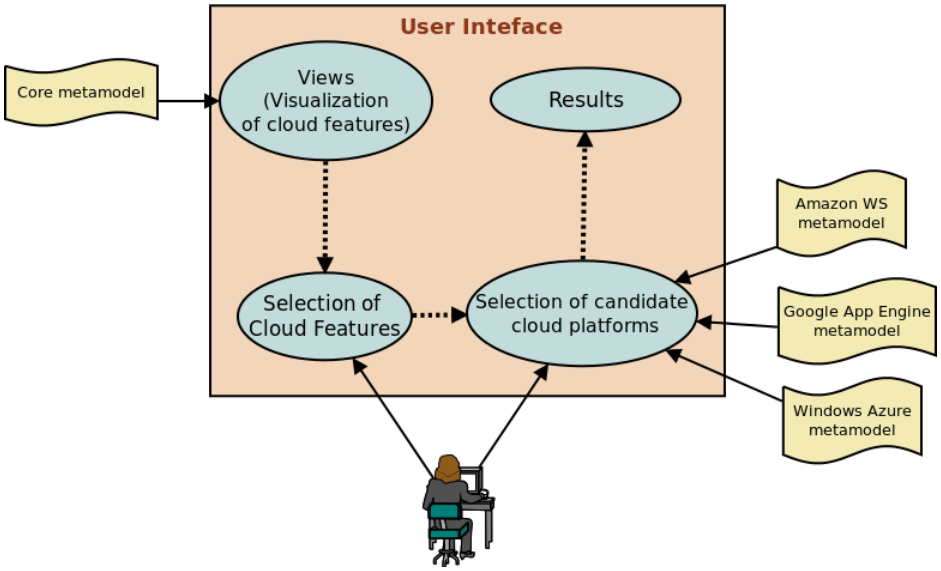

Figure 2 Cloud Target Selection tool User Interface

2.2 Technical description

The Cloud Target Selection Tool is implemented as an Eclipse plug-in. In this respect, Eclipse libraries are used in order to provide the user interface which is essential for the tool to become functional and fulfil its purpose, as a user-driven decision making engine. In addition, a profile querying mechanism has been developed, with the support of the UML2 Java library (see Section 2.2), in order to exchange information between the CloudML@ARTIST meta-models [11] and the user interface. All Java classes composing this tool are developed in order to contribute into the following tasks (see Figure 3):

1. Read the core meta-model providing all the profiles/stereotypes that are applicable on the target cloud platform meta-models.
2. Transform the information coming from task 1 and visualize it using tree-structures including checkboxes for enabling the user to make selections.
3. Obtain the user's selection through the checkboxes.
4. Transform user's selection into a UML-compatible format by forming and populating query structures specifically created for this purpose.
5. Perform queries to each of the meta-models representing the target cloud platforms.
6. Validate the results of the queries by assigning scores to each target cloud platform according to the level of compatibility with the user's requirements as set in step 4.
7. Report the results to the user.

Each of these tasks, except for the cloud features visualization (Task 2), and the report of the validation results (Task 7), are hidden from the end-user.

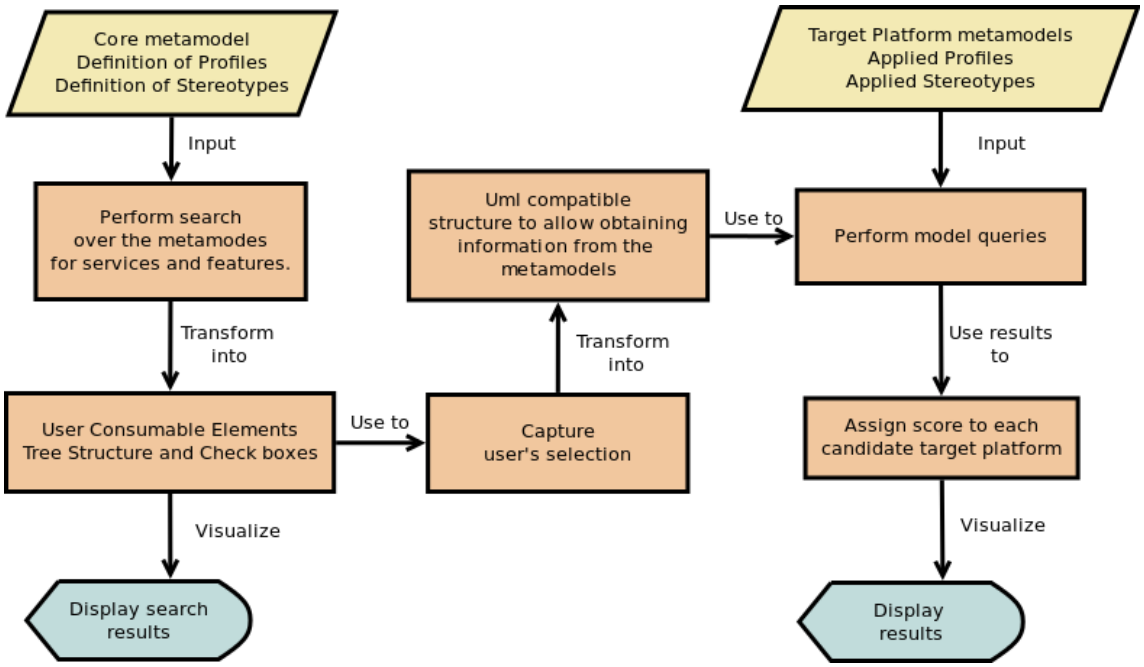

Figure 3 Sub tasks performed during the target selection process

2.2.1 Cloud Target Selection Tool Architecture

The Java classes created in the context of the Cloud Target selection tool are placed in four different packages, following a conceptual grouping process. Each package plays a different role in the architecture, as a whole. All the packages to be described are sub-folders of the general source package: “eu.artist.migration.cloudselection”.

- **UML Model Service:** This package is responsible for any action connected with UML-related tasks. Such tasks include UML file handling, model querying, memory handling (total control of loading and unloading UML resources) and transformation of the query results into forms or structures accessible by any other package with no dependencies by the UML Java library. In addition, it is responsible for making the validation of user's requirements in a match-making process between the requests to the models and the corresponding responses.
- **User Interface:** This package (including two sub-packages) contains all classes which compose the user interface, dedicated to interact with the end user. Every information exchange between the tool and the user is defined and controlled through this component.

  View Data-model: This component represents the data structure which is visualized so as to be accessed by the end user through the views. It provides a model which acts as a bridge between the user-accessible and the UML-formatted information. While it does not contain any definition of tasks to be performed, it is used to store information, which will eventually be used by every other component of the tool.

2.2.2 Components description

2.2.2.1 UML Model Service

This component is involved in any task that includes UML file handling or interaction with UML resources and model elements. More precisely, user actions that trigger classes from this package are (referring to the corresponding user actions as described in section 2.1):

Action 1: When the user demands for a view to open, this component is accessed in order to:

- Load the UML resources (R1 in Figure 4)
- Perform a search over these resources in order to find services and features to display to the user. During this search, applied instances of the stereotypes "CommonFeatures" and "Service" are to be found.
- Translate the results into "view data-model elements" (R2 in Figure 4).

Action 4: When the user asks for a validation to be made, this component:

- Gets as input all the user requests (R1 in Figure 4).
- Translates these requests into UML-compatible requests. This step requires the definition of structures for allowing different types of model queries to be generated and stored.
- Loads, one by one, the UML resources of the potential cloud target platforms. For each loaded resource, the component performs the queries resulting from the previous steps of the action, and after having obtained and stored the results, unloads, one by one, the UML resources. So for each resource the action chain is: load-validate-unload.
- Processes the query results in order to make the final suggestion.

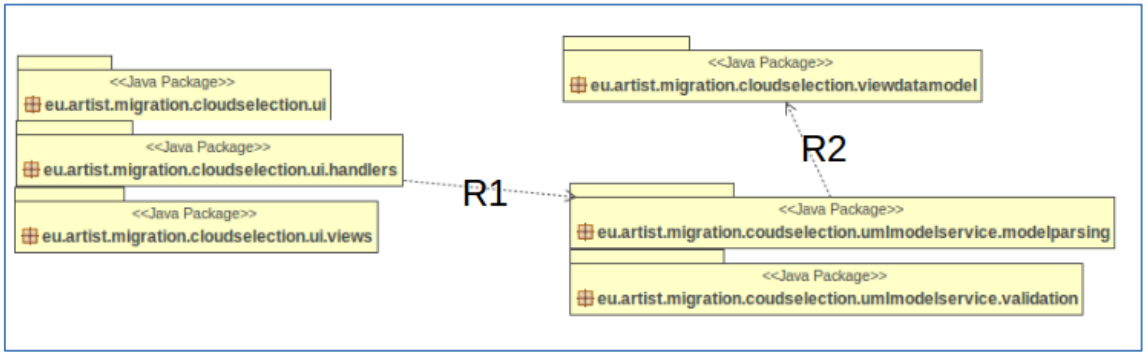

Figure 4 Package diagram with relations of UML model service component

A very important feature of the UML Model Service component is the fact that it performs the search over the core UML file, in a generic way. This means that the Java classes are unaware of the structure of the UML models to be queried in the highest possible degree. The only information hardcoded in the classes is about the name of the two stereotypes investigated ("CommonFeatures" and "Service") and about one Enumeration ("HighLevelEvaluation"), which is the datatype of some properties. This means that possible changes in the cloud meta-model will not necessarily indicate changes to the Cloud Target Selection tool. Furthermore, it provides the tool with the benefit of extensibility (in the future, more profiles can be used as input for constructing different views containing different types of features for the user to select, without affecting the implementation for the already included profiles).

Another issue to be discussed is the way the scores are assigned to each target platform, according to the query results. The score assigning process is different for the detected types of queries and answers:

- **Type1 -** High-level evaluation features queries. These queries aim at validating over features which are of type "HighLevelEvaluation". This is an enumeration allowing three values: "poor", "average", "extensive". If the user requests for such a feature then the score assignment is defined as:
  - case poor, score = 0
  - case average, score = 0.5
  - case extensive, score = 1
- **Type2 -** Boolean feature queries. These queries aim at indicating whether a feature or service exists or not. The possible answers are translated into scores as:
  - if exists, score = 1
  - if does not exist, score = 0
- **Type3 -** Multivalued element queries. These queries apply in the case of service-based evaluation (and could apply in other views too, if extended). Again they indicate whether a feature exists, but the features are grouped into service categories. For instance, in case of Infrastructure as a Service, the Storage Service may support a number of storage types such as raw storage, volume storage and block blobs. If the user selects n features of a specific service type, and the potential target platform provides m of them (m<=n) then: score = m/n

In any case, 0<=score<=1 for every single query.

Final score for every potential platform will be: $(s_1+s_2+\ldots+s_n)/n$, where s_i the score for the i_{th} query.

2.2.2.2 View Data-Model

The Data model component is used by all the other components of the tool, while it is independent from them. As shown in Figure 5, it contains the basic class which is the "ModelElement". Every other class extends it. While most classes do not have functional differences among them, the declaration of so many distinct elements is essential, in order for the tree-viewer of the graphical user interface to be aware of how to present each item, and for the UML parsing component to be aware of how to search for each one of them in the models, in case user selects it. In other words, the whole component serves as the "encoding" and "decoding" mechanism between the specific UML entities, and the data generically for user consuming.

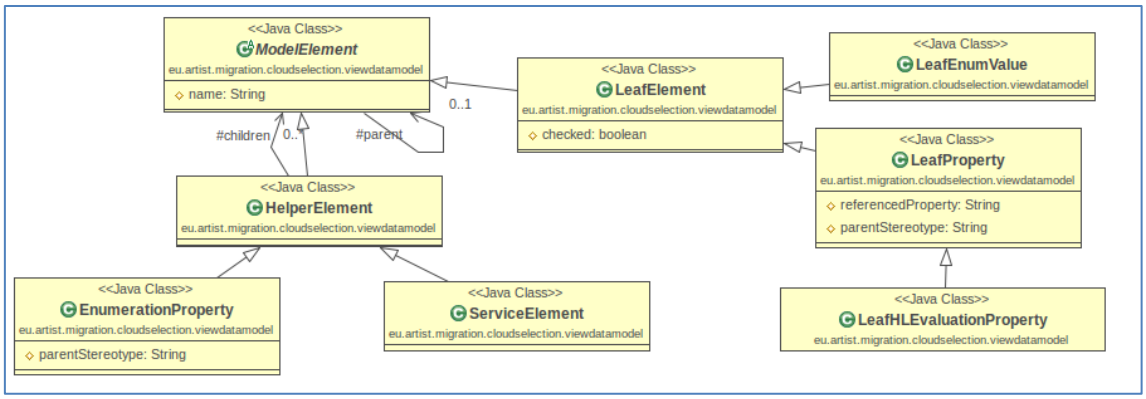

Figure 5 Class Diagram of View Data-Model

2.2.2.3 *User Interface*

The user interface has two main features: views and handlers.

Views

This component contains all the classes that handle communication between the tool and the user, which is enabled through a graphical user interface. This GUI relies mostly on the tools provided by Eclipse for plug-in development. For the time being, two very important views are available (following a twofold interpretation of the core meta-model of CloudML@ARTIST). Each one of them presents a tree-view, containing information coming from the core meta-model parsing.

- The first view (Figure 6) comes from the applied stereotype “CommonFeature”. With regard to the query types as discussed in Section 2.2.2.1, it contains features of Type1 (e.g. monitoring, support, API), Type2 (e.g. scaleup, supportsOCCI) and Type3 (e.g. Scope).
- The second view comes from all the applied stereotypes which extend the stereotype “Service” and are shown in (Figure 7). This view contains only features of Type3. However, apart from the selectable features, it exploits the view data-model in such say, so as to contain information about the grouping of the applied stereotypes into different profiles (IaaS and PaaS are UML profiles included in the core profile).

In addition to these two views, there is a third one which is responsible for presenting all the available providers. It also enables the selection of some among these providers which will be the subjects of the research to be performed later on in the tool’s usage.

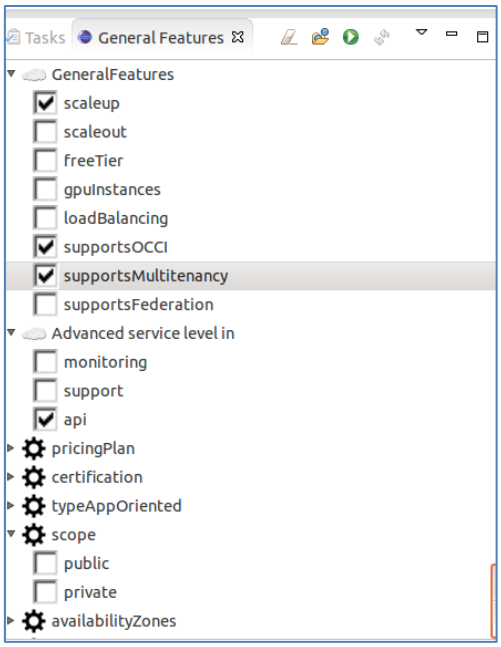

Figure 6 General Features view

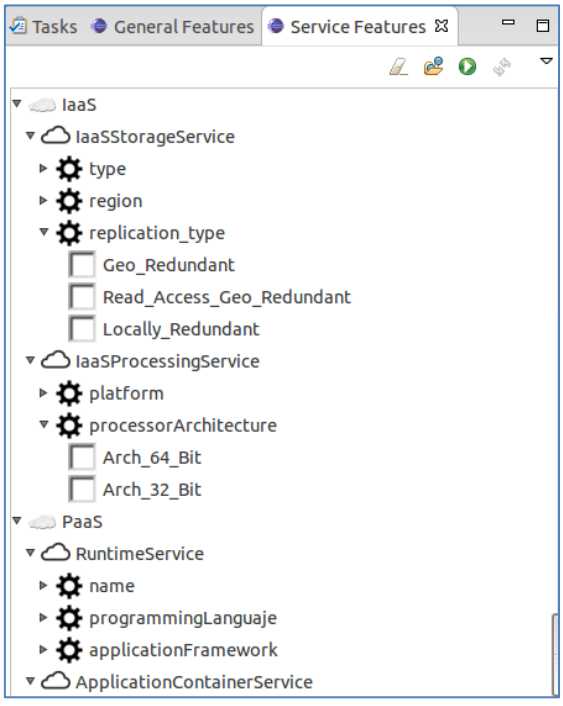

Figure 7 Service Features view

Handlers

In order to exploit the potentials provided by the view, four handlers have been designed and developed. Some of them can be used to handle events such as pressing one of the buttons appearing on the right upper corner of each view, or executing a command incorporated to the menu. The implemented handlers are:

- ProvidersFileHandler: Handles the opening of the view containing the list of the available providers.
- ValidateCommonsHandler: Initiates the process of making the validation for the selections in the "general features" view.
- ValidateServiceHandler: Initiates the process of making validation for the selections in the "Service Features" view.
- ResultsHandler: Shows the result to the user.

2.2.3 Technical specification

The Cloud Target Selection tool has been implemented as an Eclipse plugin and has been tested against Eclipse Java EE IDE, version Kepler Service Release. It requires Java v7. It extends the Eclipse IDE by implementing commands and views as well as contributing to the main menu. In addition, it uses uml2 resources[4] in order to parse UML profiles. Another plugin used is the CloudML@ARTIST plugin which is responsible for providing the metamodel on which the target selection is based. As a result, the required plug-ins are shown in Figure 8.

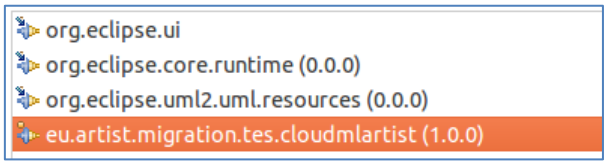

Figure 8 Plug-in dependencies for Cloud Target Selection Tool

3 Modelling Deployment in UML

General-purpose languages, such as UML, provide modelling concepts to represent software, platform, and infrastructure artefacts from different viewpoints where the deployment view is of particular relevance for specifying the distribution of software components on the targeted cloud environments. Providing extensions to UML that satisfy current cloud modelling requirements appears beneficial, especially when cloud-oriented migration scenarios [3] need to be supported where reverse-engineered UML models are tailored towards a selected cloud environment. For that reason, we have proposed the Cloud Application Modelling Language (CAML) [4][5], that enables cloud-based deployment topologies to be represented directly in UML and their refinement towards a concrete cloud environment. Features of existing cloud environments are captured by dedicated UML profiles. Thereby, a clear separation is achieved between cloud-environment independent and cloud-environment specific deployment models [6], which is in accordance with the PIM/PSM concept. In our case, the "platform" refers to the cloud environment. We developed profiles for three major cloud environments: Amazon AWS[5], Google Cloud Platform[6], Microsoft Azure[7]. Inspired from common cloud computing literature [7][8][9], recent cloud modelling approaches [10] and cloud programming approaches[8], we developed CAML's model library that facilitates developing base deployment topologies to which cloud environment profiles are applied. The benefits of realizing CAML as an internal language of UML are threefold: (i) UML provides a rich base language for the

[4] http://eclipse.org/modeling/mdt/?project=uml2
[5] Amazon AWS: http://aws.amazon.com
[6] Google Cloud Platform: http://cloud.google.com
[7] Microsoft Azure: http://azure.microsoft.com
[8] Deltacloud: https://deltacloud.apache.org and jclouds: http://jclouds.apache.org

deployment viewpoint, (ii) "cloudifying" UML models is facilitated without the need to re-model existing applications, and (iii) profiles in UML allow hiding details of cloud provider offerings from models and dynamically switching between them by (un-/re-)applying respective cloud environment profiles. CAML is considered as part of the CloudML@ARTIST [11].

3.1 CAML By-Example

To emphasize the benefits of employing UML as the host language for realizing CAML, we give an overview of UML's structural viewpoints that support representing application deployments by means of the ARTIST reference use case[9]. We take the viewpoint of the software components and their deployment. Figure 9a depicts some components of our application, an excerpt of their realizing classes and the manifestation of these components by deployable artefacts. A possible on-premise deployment for them is presented in Figure 9b. It covers instances of the two deployable artefacts and connects them to a Java-based middleware and a relational DBMS, which are in turn deployed onto a node with specified (virtual) machine characteristics. The model elements of the deployment are instances of the custom types defined in the component viewpoint, cf. Figure 9a, and the deployment viewpoint, cf. **¡Error! No se encuentra el origen de la referencia.**c, respectively. To exploit modern cloud features, deployment models need to be expressive enough to capture them. This is exactly the purpose of CAML. Because it is realized in terms of lightweight extensions to UML, CAML models are applicable to UML models and so to our modelled reference use case as depicted in Figure 9.

To demonstrate how CAML is applied, Figure 10 presents a possible deployment topology and refinement towards the Google App Engine of our introduced use case, cf. Figure 9. In a first step, we modelled the deployment topology. It consists of two automatically scaled cloud nodes and a key-value cloud storage for managing the application data in an eventually consistent way.

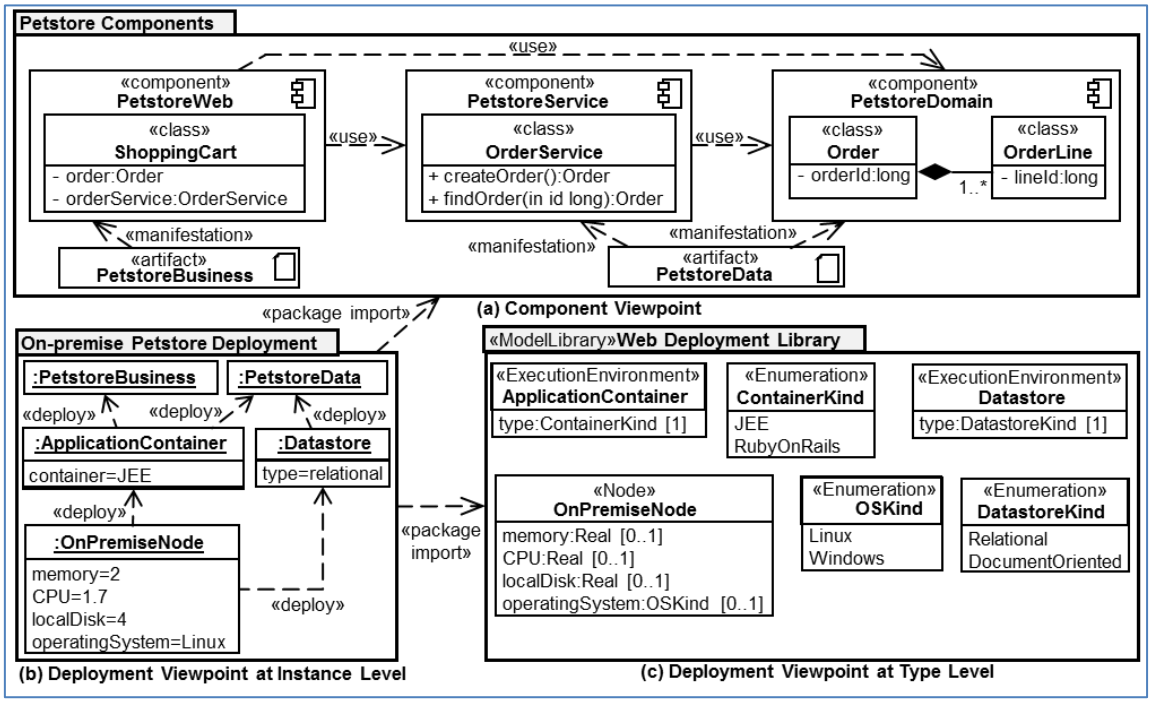

Figure 9 On-premise deployment of reference use case

[9] It is based on the Petstore Application as introduced in ARTIST deliverable D9.1 [12]

As the cloud nodes are specified as platform-level offering, we directly deployed the application components onto them. Then, in a second step, we applied the Google App Engine profile and the respective stereotypes to refine the deployment model towards concrete cloud offerings provided by the Google App Engine. As a result, the modelled cloud nodes refer to the "F1" and "F4" instance types that host a Java-based middleware. The configuration attached to these cloud nodes constrains the maximum number of idle cloud nodes. Finally, Google App Engine's key-value data-store is employed for the required cloud storage capabilities.

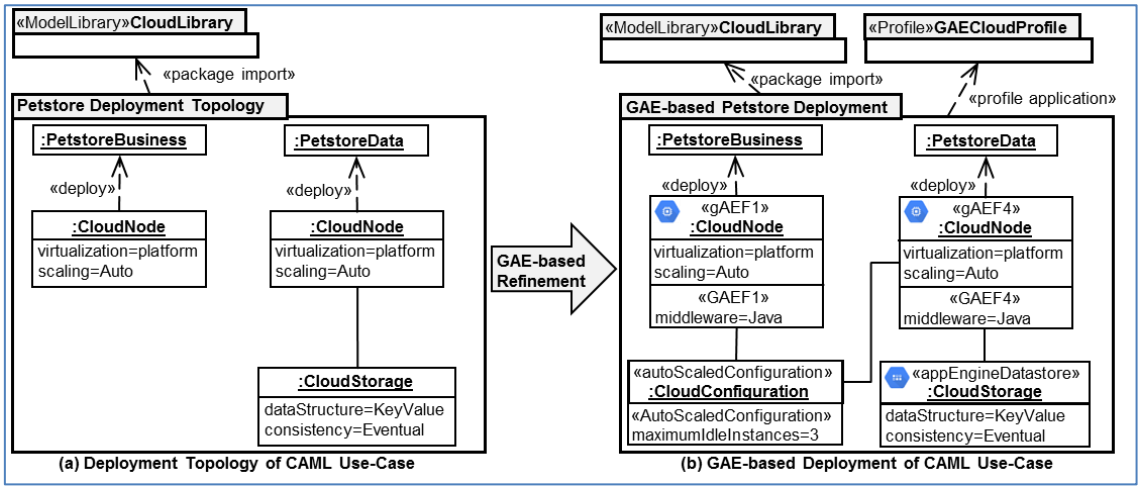

Figure 10 Reference use case deployed onto Google App Engine

3.2 Reusable Deployment Templates

As CAML is based on UML, its reuse mechanisms can be applied for cloud application deployments. This is particularly useful for providing frequently occurring deployment patterns as predefined UML templates. To show their usefulness and give first evidence of CAML's expressivity, we developed several templates as reusable deployment blueprints, most of them are based on Amazon's best practices[10]. We modelled their inherent topology with CAML's cloud library and refined them with stereotypes from the cloud profile dedicated to Amazon. To demonstrate the use of a blueprint, we show how our reference use case is bound to a template, which refers in our case to a 2-tier web architecture [9]. To reuse the predefined template, the deployable artefacts need to be bound to the template parameters. Figure 11 depicts the component viewpoint of our reference use case and the respective CAML template. It consists of two cloud nodes that refer to the "M3Medium" offering of Amazon. Their location is required to be in Europe while the operation system needs to be Linux. For reliability reasons, they are placed in different availability zones. Requests that arrive at the cloud nodes are first handled by a load balancing service, which enables a higher fault tolerance of the application. The number of running cloud nodes is automatically managed by Amazon as expressed by the scalability strategy. Only the minimum number of running cloud nodes and their adjustment is configured. Both cloud nodes are connected to a cloud storage that in turn is replicated to improve data availability. Finally, as Amazon cloud nodes operate at the infrastructure level, the required middleware for our reference application is defined. In fact, we directly reused it from the on-premise deployment given in Figure 9.

[10] Amazon Architecture Center: https://aws.amazon.com/architecture

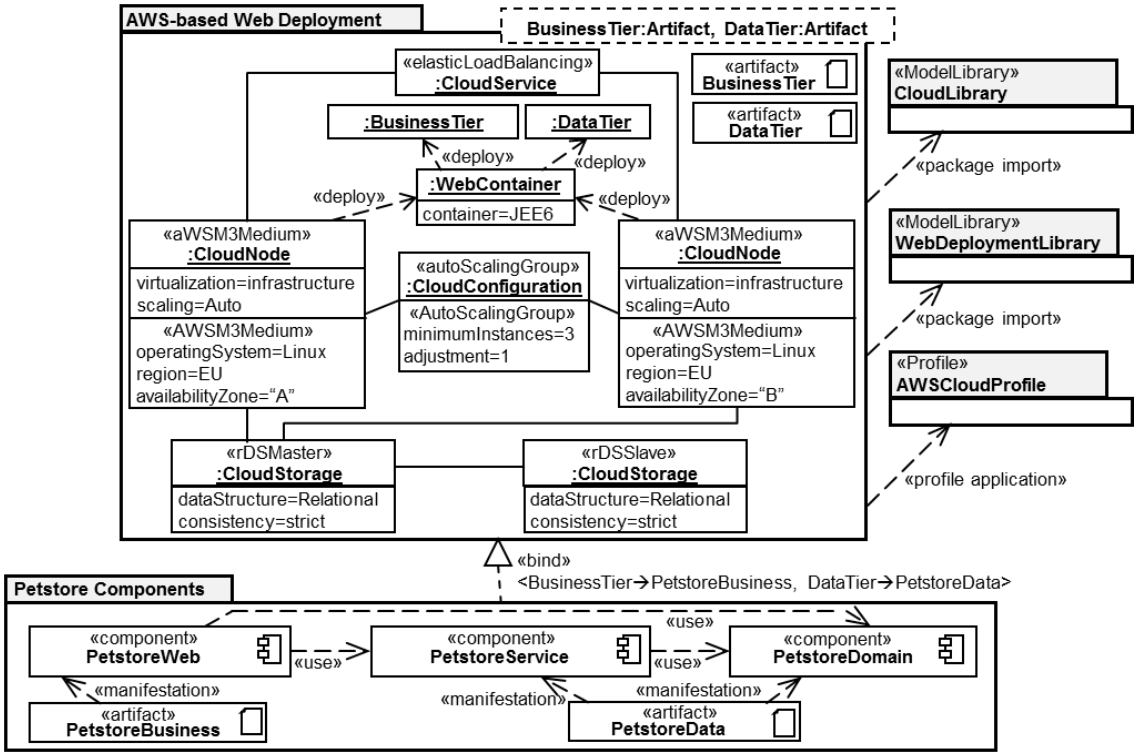

Figure 11 Reusable Deployment Template for Amazon AWS

3.3 Interoperability with Cloud Modelling Approaches and Standards

One major aspect in model-based engineering is to place models as first-class entities in the engineering process. Ideally, they should be turned into executable or interpretable artefacts. Regarding the deployment viewpoint, it appears desirable to translate the respective models into descriptors and scripts that are passed to provisioning engines for cloud environments. For instance, a Google App Engine based deployment requires specific descriptors for defining the assignment of application modules to a concrete instance type. This assignment can be derived from a CAML model as discussed in Section **¡Error! No se encuentra el origen de la referencia.**. At the same time, there are ongoing efforts in standardizing the specification of cloud-based application deployments. The recently accepted TOSCA standard aims at supporting portable cloud applications. With the notion of management plans, emerging TOSCA-compliant engines are capable to interpret such deployment topologies and initiate the provisioning of defined service templates [13]. Also, in the ModaClouds project, a provisioning engine for CloudML is developed [14]. Clearly, this is also of practical value for CAML models. For that reason, we have developed dedicated model transformations to enable interoperability between these languages.

4 Deployment Tool

4.1 Functional description

This section introduces the functional scope of the Deployment Tool and its deployment support and collects the main requirements that drive its design and development.

From a functional point of view, the techniques and tools developed in this task (in collaboration with the T9.2 CloudML specification, from the application perspective, and the T9.4 Target Generation) offer to the end-user the following features:

- Ability to **express deployment requirements and information on the modernized models** that describes the migrated application (or specific components)
  - Ability to express **deployment units** (i.e. separate deployment units corresponding to deployable components)
  - Ability to express **deployment patterns/topologies** (i.e. to specify the deployment layout of components,)
  - Ability to express both **platform and infrastructure requirements** (i.e. services or frameworks)
- Ability to **generate the deployment descriptors**, for the entire application or for individual components, for the selected target Cloud environment.
  - Deployment descriptors are editable and modifiable by end user using third-party editors (i.e. Eclipse XML Editors)
- Ability to **package deployment units**, for the entire application or for individual components, compliant with the frameworks they were developed for (platform) or with the VM specification format (infrastructure). This feature requires that compilable units have been created before using the Target Generation Tool (T9.4)
  - Deployment units can be open, browsed and modified using third party tools (i.e. Jar/Zip editors for WAR, EAR or VMWare OVF Tool, etc.).
- Ability to **generate deployment scripts** that launch the deployment process on the selected target Cloud environment. These scripts require the availability of a command-line interface provided by the SDK of the selected target Cloud environment.

4.2 Technical description

4.2.1 Deployment Tool architecture

The model-driven approach for automating Cloud deployment strategies, implemented in the Deployment tool is depicted below in Figure 12.

Figure 12 Deployment Tool process

Conceptually, this is a two-steps process. In the first step an application deployment PSM is computed for the concrete target Cloud provider. In the second step, the deployment artefacts are generated from former deployment PSM. Let's further specify these two steps separately.

The following Figure 13 depicts the steps to generate the application deployment PSM.

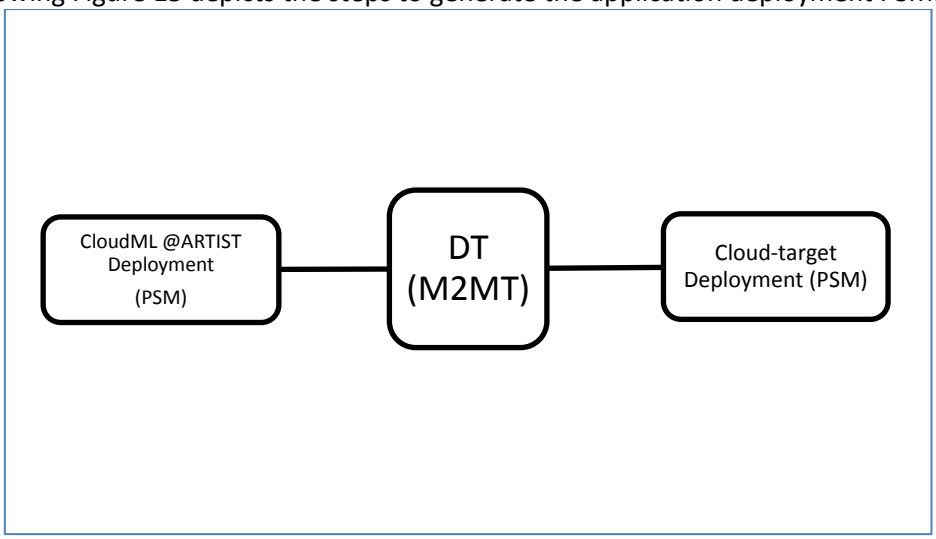

Figure 13 Deployment Tool process. Generation of the application deployment PSM

In this first step, the Deployment Tool generates a model (PSM) for deploying the application in a selected target Cloud provider. This model will contain all the information required to generate the deployment artefacts for the selected target Cloud provider. This process requires the following input elements:

- An UML deployment model (PSM) of the application. This model, apart from describing the application itself (and its constituting components), distributed in an deployment layout (i.e. UML nodes), contains additional deployment information (specified by using CloudML@ARTIST profiles) about:
  - Selected target Cloud environment for deployment.
  - Required Cloud target-specific services or frameworks (e.g. see deployment meta-model description in Appendix C) that require to be configured during the deployment.
  - Deployment requirements that can be likely communicated in a platform independent way (expressed using CloudML@ARTIST, see PI meta-model in Appendix C). If the user needs to specify platform specific deployment requirements, she can do it, using the CloudML@ARTIST profile available for the selected target Cloud environment.

As depicted in the **¡Error! No se encuentra el origen de la referencia.**, this model is fed into the Deployment Tool (conceptualized in the picture as a M2M deployment pattern or a set of M2M deployment patterns). The purpose of this first step is to produce a *platform specific deployment model* for the application that personalises all the deployment requirements according to the selected target Cloud environment. In order words, a set of M2M transformations convert an instance of UML deployment meta-model (annotated with CloudML@ARTIST profiles) into one or more instances (compliant to one or more cloud-target-specific meta-models) that describe the deployment of the application into the concrete cloud-target offering.

In the second step (see Figure 14 below), the Deployment Tool applies a set of M2T transformations to the application deployment model (PSM) to obtain the target specific deployment descriptors. Additionally, the Deployment Tool can pack these descriptors with the migrated compilation units (provided as input) to generate deployment units (not depicted in the picture). Besides, a similar technique based on M2T transformations can be used to generate target-specific deployment scripts, which are functionally described in previous sections.

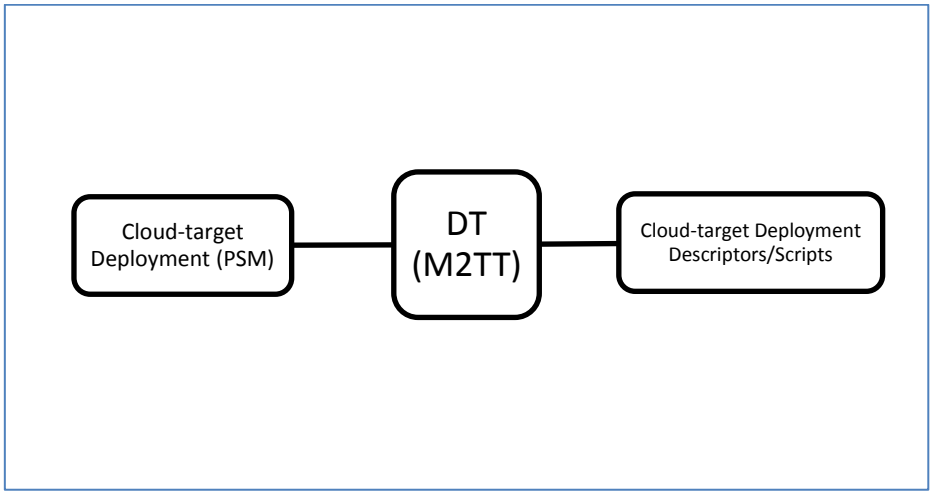

Figure 14 Deployment Tool process: Artefacts generation.

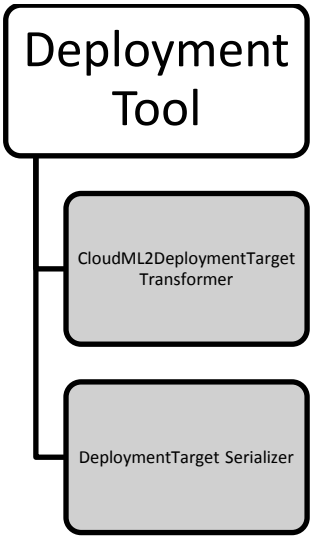

Figure 15 Deployment Tool components

The main Deployment Tool components are depicted in Figure 15. The CloudML2Deployment Target Transformer processes the first step of the deployment descriptors generation described in Figure 13, while the Deployment Target Serializer obtains the serialized deployment descriptors as described in Figure 14. Both components exchange the cloud-target-specific deployment models obtained in the first step.

4.2.2 Components description

Generating deployment scripts for Google App Engine and Microsoft Azure from a deployment model expressed in CloudML@ARTIST/CAML, requires overcoming the different encodings and

providing a conceptual mapping of the various concepts. The latter is the basis for implementing a transformation to automate the generation of pertinent deployment scripts. First, we describe how we bridge the different encodings by providing an Ecore-based metamodel as a bridging technology (see Section 4.2.2.1). Then, we introduce the mappings we have implemented in terms of model-to-model transformations where the target metamodels of them are the bridging metamodels (see Section 4.2.2.2).

4.2.2.1 Metamodels for Bridging the Technical Spaces

To generate deployment scripts for Google App Engine and Microsoft Azure from a deployment model, we are confronted with two technical spaces: *Modelware* and *XMLware*. Basially, the deployment scripts need to be encoded in the correct format, which is XML, whereas the common format for encoding models is XMI as standardized by the OMG. Moreover, they need to conform to the respective XML schemas provided by both Google App Engine and Microsoft Azure. These XML schemas provide all the concepts to which correspondences from the concepts of CloudML@ARTIST/CAML need to be identified and implemented in terms of transformations. Directly implementing a transformation that produces deployment scripts in XML format would require overcoming two challenges at once. Hence, we advocate a two-step approach, where in a first a model-to-model transformation is applied to translate between concepts of CloudML@ARTIST/CAML and the target cloud environment. Then, in a second step, the produced model of the model-to-model transformation is translated into the encoding expected by the cloud environment. To achieve such a two-step approach an Ecore-based metamodel is used as a bridge between the two technical spaces [15][16]. In fact, an Ecore-based metamodel can be automatically generated from an XML schema while the parser and printer to translate between the technical spaces is provided by EMF. Figure 16 summaries the use of Ecore-based metamodels as a bridging technology. The generated metamodel can be used as the target for model-to-model transformations. This is exactly the approach we followed to implement the transformations of our deployment tool as discussed in Section 4.2.2.2.

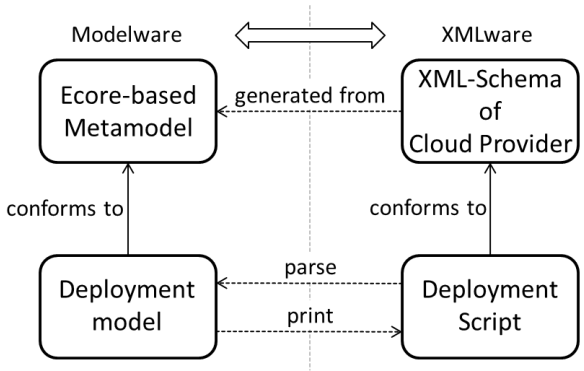

Figure 16 Ecore-based metamodels as a bridge between Modelware and XMLware

4.2.2.2 CloudML2DeploymentTarget Transformer

This section describes the M2M transformer that converts an UML deployment model into a set of cloud-target-specific (Google App Engine, Microsoft Azure) deployment models (one of each available cloud-target-specific meta-models), which are finally serialized by the Descriptor Serializer (see Section 4.2.2.1).

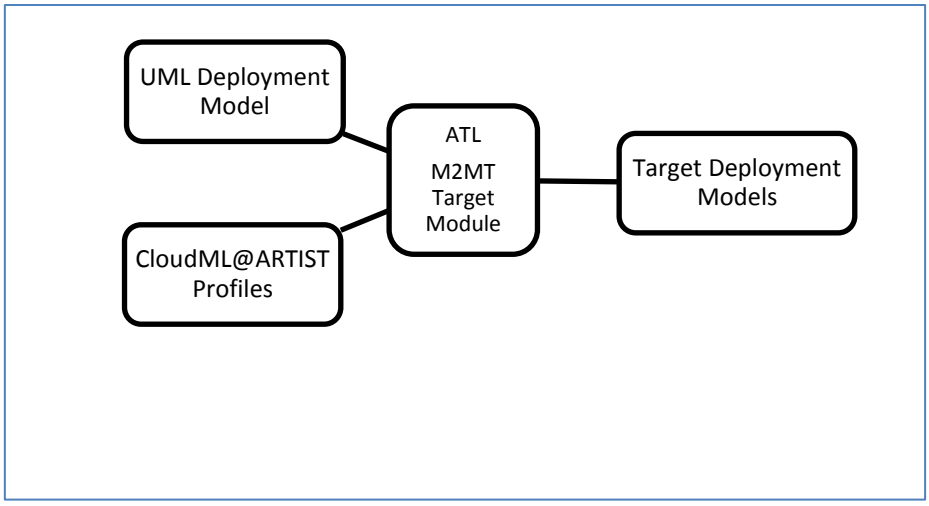

Figure 17 CloudML2Deployment Transformer

Figure 17 depicts the process performed by CloudML2DeploymentTarget Transformer components, in terms of inputs and outputs of the ATL M2MT target module.

This module takes as input:

- The deployment model designed by the user, which is an instance of the UML meta-model, and has been annotated with stereotypes taken from the CloudML@ARTIST DSL.
- The CloudML@ARTIST profiles that are manage by the module, particularly those providing a deployment vocabulary.

This module produces, as output:

- The specific deployment models for the selected cloud target. These models are concrete instances of the cloud-target-specific meta-models. In particular, Google App Engine requires one concrete deployment descriptor for Java-based applications (i.e. appengine-web), whose concrete model is an instance of the Google App Engine AppEngineWeb metamodel. Microsoft Azure requires different service descriptors: configuration, definition and description, whereby three different models (as instances of Microsoft Azure service configuration, definition and description) are created.

In practice, different ATL modules are specialized to create one single output deployment descriptor model. As such, to generate the different Microsoft Azure descriptors, different ATL modules are invoked in sequence.

The following code snippet contains the ATL module declaration for the CloudM2GAE transformation, which describes the input and out signature of the module:

```
1  -- @nsURI AEWMM=http://appengine.google.com/ns/1.0/appengine-web-app
2  -- @nsURI UMLMM=http://www.eclipse.org/uml2/4.0.0/UML
3
4  -- GAEP: Google App Engine Profile
5  -- CMLCP: CloudML@ARTIST Core Profile
6  module caml2gaeweb;
7  create AEWM : AEWMM from UMLM : UMLMM, GAEP : UMLMM, CMLCP : UMLMM, SECP : UMLMM;
8
```

As seen in the module declaration, this M2MT creates a Google App Engine appengine-web model from an UML model, using the CloudML@ARTIST profile for Google App Engine, the core profile and the security profile.

The following ATL rule defines the mapping between an CloudNode instance (of type GAE) in the CloudML@ARTIST input deployment model and the AppengineWebAppType element in the output cloud-target-specific deployment model:

```
-- create an AppengineWebApp / Module for each GAEInstanceType
rule GAEInstanceType2AppengineWebApp {
    from s: UMLMM!InstanceSpecification (s.isGAEInstanceType())
    to t: AEWMM!AppengineWebAppType (
        -- TODO: check what should be the name
        application <- s->debug('CloudNode').getApplicationName(),
        module <- s.getModuleName(),
        version <- '1.0',
        sessionsEnabled <- false,
        sslEnabled <- if (s.declaresSecurityConcerns()) then true else false endif,
        threadsafe <- s.getThreadSafe(),
        asyncSessionPersistence <- thisModule->createAsyncSessionPersistence(s),
        inboundServices <- if (s.declaresMailServiceInConfiguration())
            then thisModule->createInboundServices() else OclUndefined endif,
        systemProperties <- let properties:Collection(OclAny) = s.getSystemProperties() in
            if not properties.oclIsUndefined() and not properties.isEmpty()
            then thisModule->createSystemProperties(properties) else OclUndefined endif,
        instanceClass <- if (s.declaresScaling()->debug ('declares scaling'))
            then s.getInstanceClass() else OclUndefined endif,
        basicScaling <- if (s.declaresBasicScaling()->debug ('declares basic scaling'))
            then thisModule->createBasicScaling(s.getBasicScaling()) else OclUndefined endif,
        automaticScaling <- if (s.declaresAutoScaling()->debug ('declares auto scaling'))
            then thisModule->createAutoScaling(s.getAutoScaling()) else OclUndefined endif
    )
    do {
        thisModule.DocumentRoot.appengineWebApp <- t;
    }
}

```

Similar ATL M2MT modules have been created to generate cloud-target-specific models from CloudML@ARTIST, for Microsoft Azure service deployment descriptors: service configuration, service definition and service description.

When some elements of the cloud-target-specific meta-models cannot be derived from the CloudML@ARTIST, default values have been set up, based on the analysis of the specification. It is up the Deployment Tool user to modify these default values when it is required.

5 Delivery and usage

5.1 Package information

5.1.1 Cloud Target Selection Tool

Figure 18 shows file structure of the plug-in. The contained files are:

- **src -** All java files included
- **icons -** Image files accessed at runtime
- **META-INF -** The plug-in manifest file
- **build.properties -** Defines all properties needed to run the plugin
- **plugin.xml -** Description of extending the eclipse platform
- **Readme.txt -** Installation and usage instructions

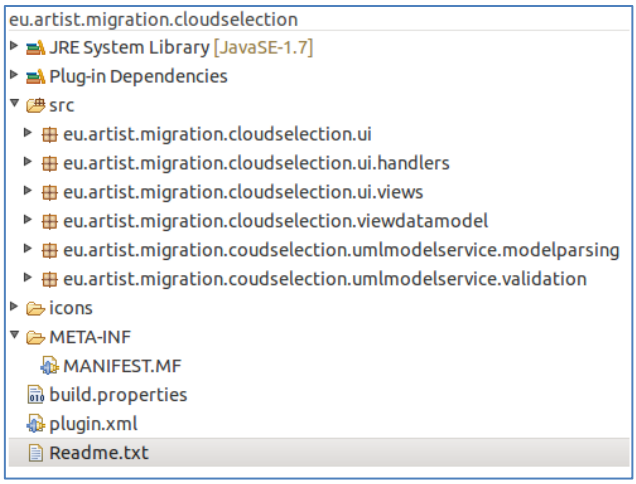

Figure 18 Package structure

5.1.2 Deployment Tool

The Deployment Tool components are packaged either as Eclipse plugins or Eclipse projects. These plugins are shown in the next Figure 19:

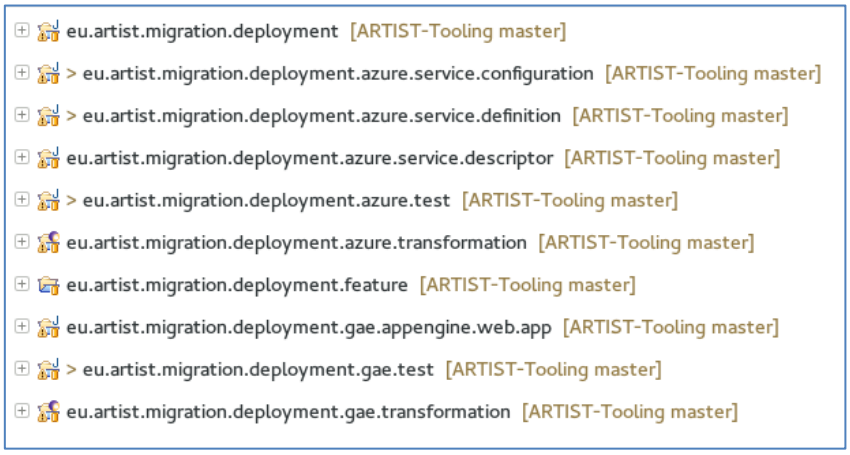

Figure 19 Deployment tool sub-projects

- *eu.artist.migration.deployment* project implements the UI of the Deployment Tool and its contributions to the Eclipse workbench. It launches the Deployment Tool generators selected by the user on a concrete deployment model.
- *eu.artist.migration.deployment.azure.service.configuration* provides the Microsoft Azure service configuration metamodel, its Java entities and EMF-based helper classes to serialize the Azure service configuration descriptor (*.cscfg).
- *eu.artist.migration.deployment.azure.service.definition* provides the Microsoft Azure service definition metamodel, its Java entities and EMF-based helper classes to serialize the Azure service definition descriptor (*.csdef).
- *eu.artist.migration.deployment.azure.service.description* provides the Microsoft Azure service description metamodel, its Java entities and EMF-based helper classes to serialize the Azure service descriptor script (*.ps).
- *eu.artist.migration.deployment.azure.transformation* provides Java helpers and ATL M2M transformations to migrate from deployment model instances of

CloudML@ARTIST to the Azure service configuration, definition, description metamodels.

- *eu.artist.migration.deployment.features* packages all Deployment Tool plugins as a single Eclipse feature.
- *eu.artist.migration.deployment.gae.appengine.web.app* provides the Google App Engine web app definition metamodel, its Java entities and EMF-based helper classes to serialize the Google App Engine web application descriptor (appengine-web.xml).
- *eu.artist.migration.deployment.gae.transformation* provides Java helpers and ATL M2M transformations to migrate from deployment model instances of CloudML@ARTIST to the Google App Engine web application definition metamodel.

5.2 Installation instructions

The Cloud Target Selection Tool plug-in has been tested on of Eclipse Kepler SR2. Java v7 is required. Before installing this tool, the user should download and install the UML2 plugin as well as the CloudML@ARTIST plugin[11]. The next step is to copy the Cloud Target Selection Tool plugins on the "dropins" folder of the Eclipse installation.

The Deployment Tool is bundle as a zip file that contains the Deployment Tool feature and its plugins. In order to install them, just unzip DT.zip into the Eclipse folder and start Eclipse. Alternatively, Deployment Tool can be installed through the ARTIST update site. Go to "Help->Install New Software", select ARTIST in the "Work with" combo. If ARTIST update site is not available, click on "Add->Archive" and browse your local file system to locate the ARTIST update site you have downloaded from the ARTIST web site.

Once the site has been loaded, in the ARTIST category, select "Deployment Tool" and install it.

5.3 User Manual

5.3.1 Cloud Target Selection Tool

In order to ensure the success of the tool's installation, the user should try selecting: Window > Show View > Other...

In the opening dialog with the available views, there should be a new category added under the label: Cloud Target Selection. There the user can select between the two available (or more, upon extension) views. Double selection is also feasible. The selected views will open but nothing will be displayed. Having managed to get to this point means that the installation has been successful and the user can make the following steps in order to make use of the plug-in:

1. Go to one of the open views. There, on the upper right corner a toolbar appears, containing a set of icons each one of them representing a specific action, which is shown simply by rolling the mouse over each icon:

[11] D7.2.3 Cloud services modelling and performance analysis framework, ARTIST EU Project

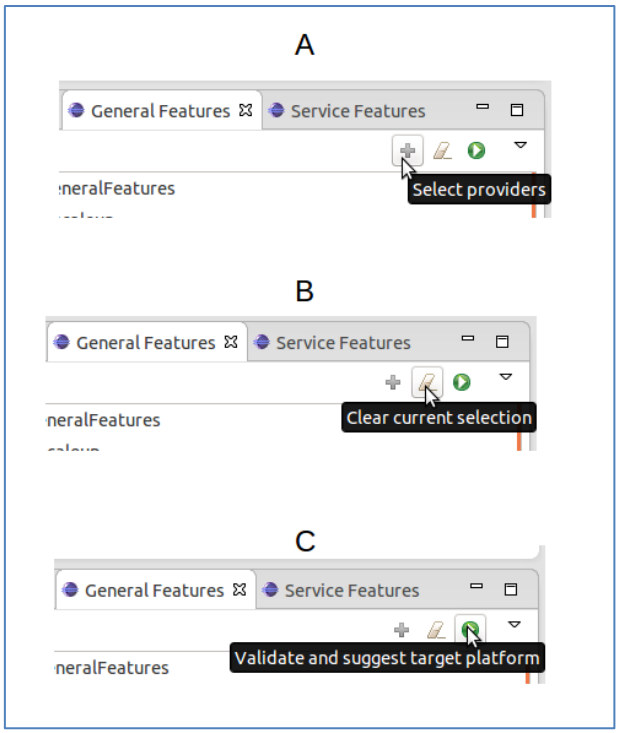

Figure 20 Toolbar of the views

2. By selecting the first icon (Figure 20A) a view will appear showing the list of supported providers to select from. These will be the candidate target platforms for the cloud target selection process.
3. The user can select any of the shown items which represent services and cloud features. The eraser button (Figure 20B), can seem helpful by unselecting every object and clearing the view.
4. When the feature selection process is finished, the validation process can begin by pressing the "play" button (Figure 20C). The validation will take place among the selected providers. If the user hasn't made a selection, all the supported providers will be checked (default configuration).
5. The results will appear on screen as scores for each of the selected providers.
6. Steps 2 to 5 can be repeated any number of times the user desires (with different input in each repetition) in order to gain a satisfying insight about the offerings of each cloud platform.

5.3.2 Deployment Tool

Deployment Tool can be used on deployment model instances of the CloudML@ARTIST. Figure 22 and Figure 23 below show deployment models for DEWS and LoB use cases respectively. These models can be created by hand using any Eclipse UML visual editor compatible with UML2 Ecore, such as Papyrus, by applying CloudML@ARTIST deployment meta-model and specific Cloud deployment profiles.

Browse your workspace, on the Navigation or Package Explorer view, and locate the deployment model you want to generate deployment descriptors from. Right-click and select the "Deployment Tool->Generate Deployment Descriptors" entry in the pop-up contextual menu (see Figure 21). A Deployment Tool dialog appears.

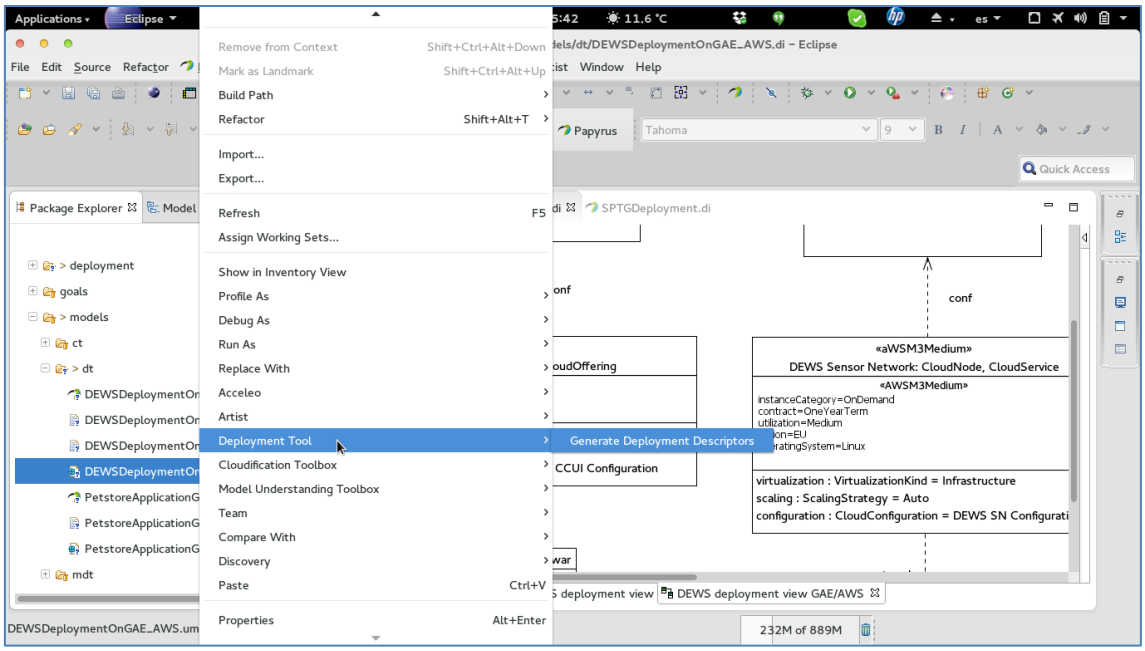

Figure 21 Deployment Tool contextual menu

In the dialog (see Figure 24), select the Cloud deployment target, that is, the Cloud offering you want to generate the descriptors for. Optionally, you can specify the target location where to place the generated descriptors. If so, click on “Browse” button and select the target project in the pop-up project selection dialog. When finished, accept the dialog to generate the descriptors. After few seconds required descriptors are generated and a modal dialog informs the user about the location of those files.

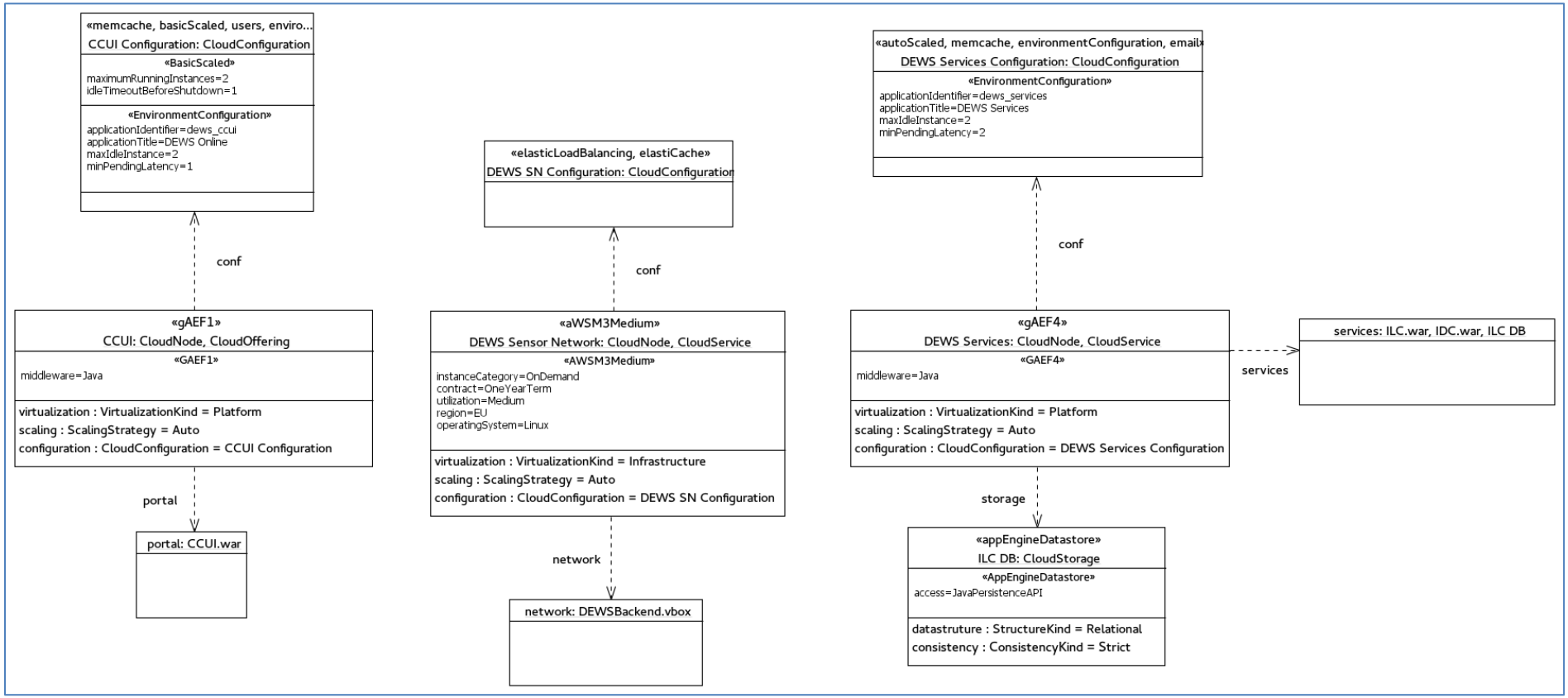

Figure 22 Deployment model for DEWS use case

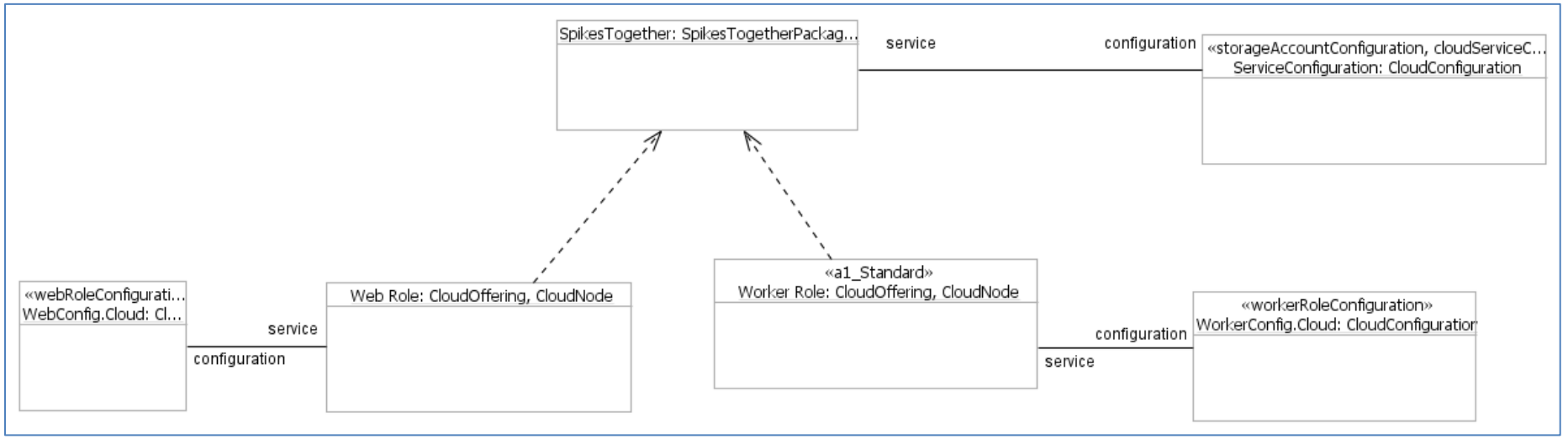

Figure 23 Deployment model for LoB use case

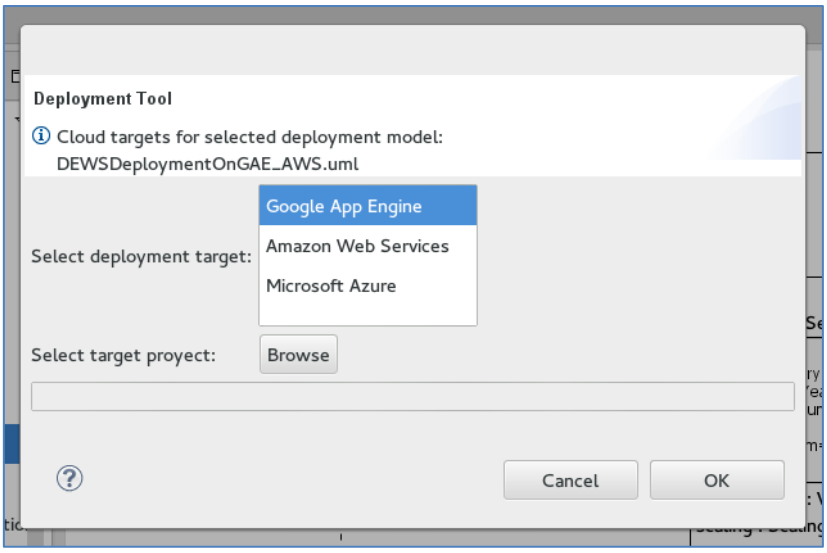

Figure 24 Deployment Tool Dialog

As an example, next figure (Figure 25) shows the generated Google App Engine descriptors for each module in the DEWS deployment model. For each module, one customized "appengine-web.xml" descriptor is generated.

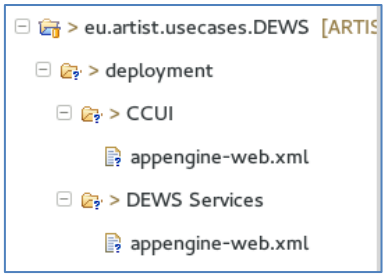

Figure 25 GAE generated deployment descriptors for each module of DEWS use case

Figure 26 shows the generated Google App Engine descriptor for the CCUI module.

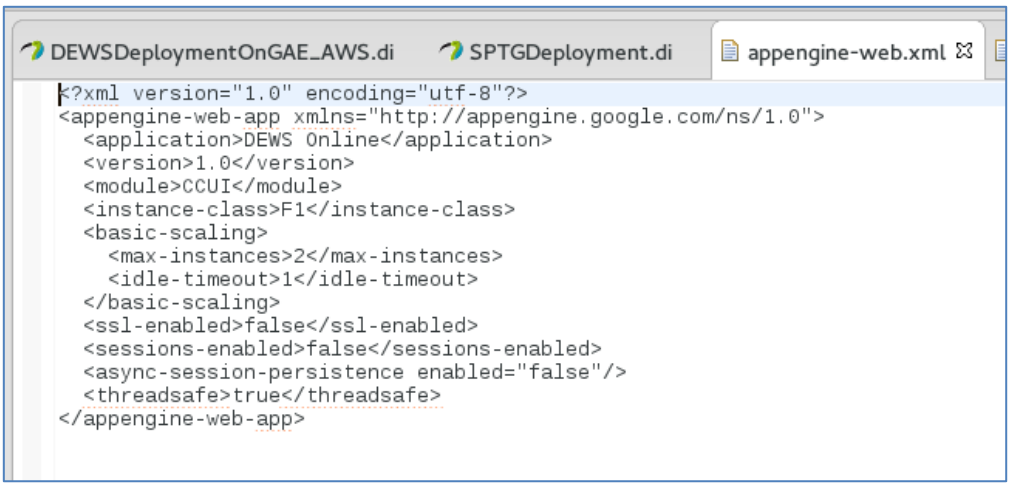

Figure 26 GAE deployment descriptor

Similarly, Figure 27 shows the Microsoft Azure generated deployment descriptors and scripts for the LoB deployment model. Two deployment descriptors (i.e. service configuration and definition) and one script (i.e. service description) are generated.

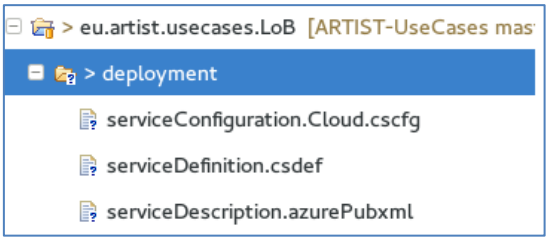

Figure 27 Azure generated deployment descriptors and scripts for each module of LoB use case

Figure 28, Figure 29 and Figure 30 show the generated Microsoft Azure service definition, service configuration descriptors and the service description script, respectively.

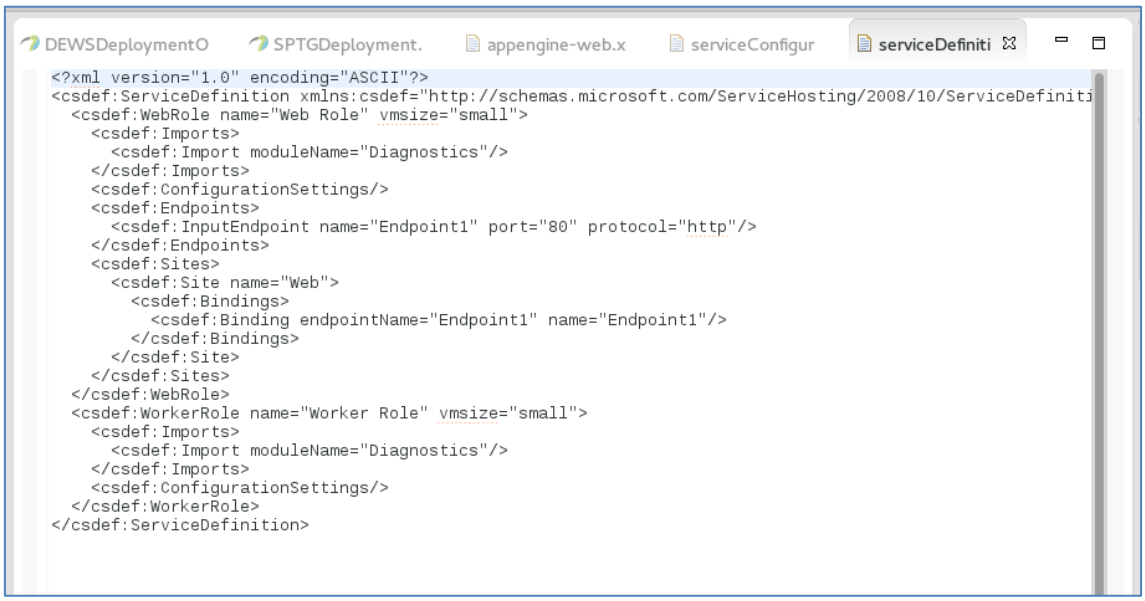

Figure 28 Azure service definition descriptor

Figure 29 Azure Service configuration descriptor

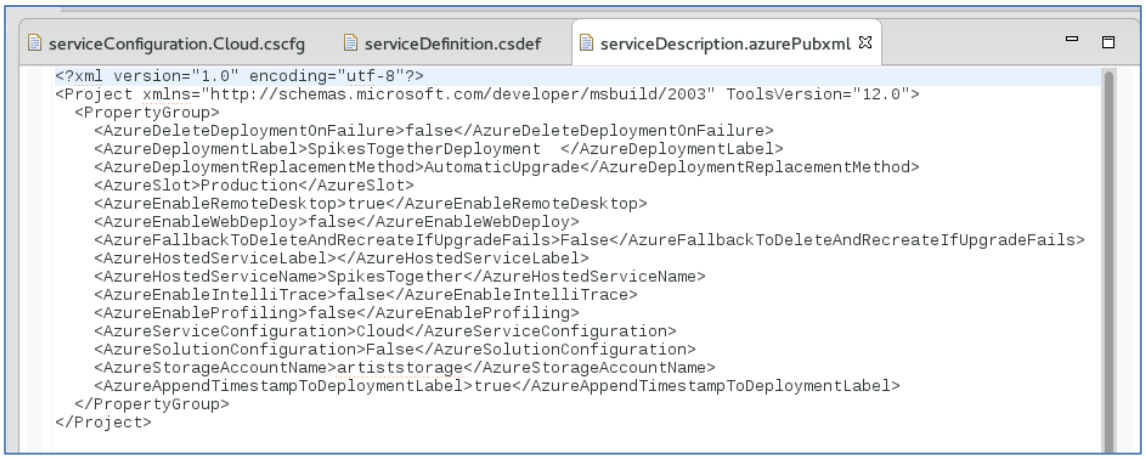

Figure 30 Azure service description

5.4 Licensing information

Both Cloud Target Selection Tool and the Deployment Tool are released under the Eclipse Public License (EPL)[12], which is a known as a "commercial-friendly" open source license. This should facilitate the future potential reuse and integration of these tools by external partners.

5.5 Download

The sources of the Cloud Target Selection Tool and the Deployment Tool are available in the public Github ARTIST repository[13] at the following location:

source/Tooling/migration/modernization/deployment

[12] https://www.eclipse.org/legal/epl-v10.html
[13] https://github.com/artist-project/ARTIST

6 Conclusions

This document introduces the theoretical, functional and technical framework for the ARTIST deployment strategies to the Cloud, which enables the specification of the application deployment requirements, the selection of suitable Cloud target environments and the automation of the deployment support. In the document, we elaborate on the scope and motivation for model-driven automated deployment strategies, in the context of the ARTIST methodology, during the modernisation activity of the migration phase. We elaborate the functional scope of the modelling language, the Cloud Target Selection Tool and the Deployment Tool, focusing on the model-driven specification of deployment models, the selection of the Cloud target environment and the generation of required deployment descriptors. For practical reasons, we restrict the deployment support for those target Cloud providers required by ARTIST use cases, namely Google App Engine and Microsoft Azure, although the proposed model-driven automated deployment approach is generic enough as to be easily extended to support any other Cloud infrastructure or platform provider.

We have analysed and reported on similar or related model-driven deployment strategies conducted in the scientific research (see APPENDIX A). On the basis of these findings and aligned to other ARTIST techniques on model-driven modernisation of non-cloud-compatible applications, we have proposed in this document a two-stages model-driven approach that makes intensive usage of CloudML@ARTIST modelling support and both M2M and M2T transformation techniques, in order to generate the artefacts required to deploy a “cloudified” application (see Section 4.2.1).

We have also analysed in detail the current deployment patterns and frameworks supported by the target ARTIST Cloud providers (see APPENDIX B), identifying their main deployment concepts and entities. Based on this analysis, we have created a platform independent (PI) meta-model (see Appendix C), which describes the main deployment concepts and entities and their relationships, both from the Cloud provider and application owner perspective, regardless of any platform specific concern. Using this PI meta-model, we have instantiate platform domain models for each of the three ARTIST selected target Cloud providers (see Appendix C).

This PI meta-model identifies an information model for Cloud deployment. This information model has influenced the development of the CloudML@ARTIST, in order to enable ARTIST to provide model-driven automated deployment support (see section 2). In this regards, we have identified entities on the deployment meta-model already included in the CloudML@ARTIST and other entities that should be included in future releases of this profile.

We have described a generic and wide enough modelling approach to specify Cloud deployment requirements and specifications at model level, and their concretization for specific Cloud offerings, making use of reusable deployment templates (see Section 3.2)

We have also provided the functional and technical specifications of the two ARTIST tools supporting the deployment to Cloud: the Cloud Target Selection Tool (see section 2) and the Deployment Selection Tool (see Section 4). The former relies on the Cloud specific profiled meta-models shipped within CloudML@ARTIST. Therefore, by adding new CloudML@ARTIST meta-models describing other Cloud providers, the Cloud Target Selection Tool will be seamlessly extended to support them as well. The latter can be easily extended as well, with little effort, to generate deployment descriptors for additional Cloud offerings, since its approach is generic enough and relies on the homogenous modelling support provided by CloudML@ARTIST.

7 References

[1] G. Baryannis and P. Garefalakis, “Lifecycle management of service-based applications on multi-clouds: a research roadmap,” in MultiCloud conf., 2013, pp. 13–20.

[2] Bergmayr, A., Grossniklaus, M., Wimmer, M., Kappel, G.: Cloud Modelling Languages by Example, In: SOCA (2014)

[3] Bergmayr, A., Bruneliere, H., C´anovas Izquierdo, J.L., Gorroñogoitia, J., Kousiouris, G.,Kyriazis, D., Langer, P., Menychtas, A., Orue-Echevarria Arrieta, L., Pezuela, C., Wimmer, M.: Migrating Legacy Software to the Cloud with ARTIST. In: CSMR (2013)

[4] D9.2 – Modelling languageand editor for defining target specifications ARTIST Project

[5] Bergmayr, A., Troya, J., Neubauer, P., Wimmer, M., Kappel, G.: UML-based Cloud Application Modelling with Libraries, Profiles and Templates. In: CloudMDE Workshop @ MoDELS (2014)

[6] Ardagna, D., Nitto, E.D., Mohagheghi, P., Mosser, S., Ballagny, C., D’Andria, F., Casale, G., Matthews, P., Nechifor, C.S., Petcu, D., Gericke, A., Sheridan, C.: MODAClouds: A Model-Driven Approach for the Design and Execution of Applications on Multiple Clouds.In: MISE Workshop (2012)

[7] Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: A View of Cloud Computing. CACM 53(4) (2010)

[8] Badger, M.L., Grance, T., Patt-Corner, R., Voas, J.M.: Cloud Computing Synopsis and Recommendations. Tech. rep., NIST Computer Security Division (2012)

[9] Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns - Fundamentals to Design, Build, and Manage Cloud Applications. Springer (2014)

[10] Bergmayr, A., Grossniklaus, M., Wimmer, M., Kappel, G.: Cloud Modelling Languages by Example, In: SOCA (2014)

[11] D7.2.1 - Cloud services modelling and performance analysis framework, ARTIST Project

[12] D9.1 State of the art in modelling languages and model transformation techniques, ARTIST Project

[13] Binz, T., Breitenb¨ucher, U., Kopp, O., Leymann, F.: TOSCA: Portable Automated Deployment and Management of Cloud Applications. In: Advanced Web Services (2014)

[14] Ferry, N., Song, H., Rossini, A., Chauvel, F., and Solberg, A.: CloudMF: Applying MDE to Tame the Complexity of Managing Multi-Cloud Applications. In: UCC (2014)

[15] Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. In: Proc. SLE. pp. 374–383 (2010)

[16] Bergmayr, A., Grossniklaus, M., Wimmer, M., Kappel, G.: JUMP-From Java Annotations to UML Profiles. In: Proc. MoDELS. pp. 552-568 (2014)

[17] G. Edwards, G. Deng, D. C. Schmidt, A. Gokhale, and B. Natarajan, “Model-driven configuration and deployment of component middleware publish/subscribe services,” in GPCE conf., 2004, vol. 3286, pp. 337–360

[18] I. Manolescu, M. Brambilla, S. Ceri, S. Comai, and P. Fraternali, “Model-driven design and deployment of service-enabled web applications,” ACM Trans. Internet Technol., vol. 5, no. 3, pp. 439–479, Aug. 2005.

[19] S. Ceri, P. Fraternali, R. Acerbis, A. Bongio, S. Butti, F. Ciapessoni, C. Conserva, R. Elli, D. Elettronica, P. Milano, and P. L. Da Vinci, “Architectural Issues and Solutions in the Development of Data-Intensive Web Applications,” in VLDB conf., 2003.

[20] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl, “Variability modelling to support customization and deployment of multi-tenant-aware Software as a Service applications,” in PESOS workshop, 2009, pp. 18–25.

[21] Czarnecki, K., and U. Eisenecker, Generative Programming: Methods, Tools, and Applications, , Reading, MA, USA, Addison-Wesley, pp. 864, 2000

[22] Software Product Lines: Practices and Patterns. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[23]N. Ferry, A. Rossini, F. Chauvel, B. Morin, and A. Solberg, “Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems,” in CLOUD conf., 2013, pp. 887–894.
[24]A. Papaioannou and K. Magoutis, “An Architecture for Evaluating Distributed Application Deployments in Multi-clouds,” in CloudCom conf., 2013, pp. 547–554.
[25]C. Quinton and N. Haderer, “Towards multi-cloud configurations using feature models and ontologies,” in MultiCloud conf., 2013, pp. 21–26.
[26]A. Gunka and H. Kühn, “Moving an Application to the Cloud – A n Evolutionary Approach,” pp. 35–42, 2013.
[27]D4.3.2 MODACloudML IDE. MODACloud Project
[28]CloudML@ARTIST to MODACloudML translation. ARTIST internal report.

APPENDIX A: Analysis of the state of the art

The deployment of web applications using MDE techniques has been tackled by several works in the last years. For instance, in [17], authors propose a model-based approach to configure and deploy publish/subscribe services in QoS-enabled component middleware based on CORBA. The proposal defines the language EQAL (Event QoS Aspect Language), which is used to configure and deploy services in QoS-enabled component middlewares. Thus, this work is coupled to a particular technology (publish/subscribe middleware in CORBA) and requires developers to define the EQAL definition to deal with their particular applications. Instead, we propose a more generic approach dealing with CloudML and particularizing for a set of concrete Cloud providers. Furthermore, our deployment process relies on a modernized application and a set of requirements which allows the generation of deployment definitions.

The works presented in [18] and [19] describe a high-level language and methodology for designing and deploying Web applications using Web services. The language allows designing web applications and web services as well as their deployment. In the former case, the deployment is done in the WebRatio[14] Architecture whereas the latter can be deployed in any web service container. As it can be seen, this approach is mainly focused on web service architectures whereas we are targeting a broader deployment context considering different Cloud vendors.

An approach for the deployment of multitenant-aware Software-as-a-Service (SaaS) applications is presented in [20], where feature models are proposed to model SaaS application configurations and to generate the corresponding deployment scripts. Feature models [21] are mainly used in software product lines [22] to define a family of products. A family of products has different variation points, which allow the definition of a configuration for a particular product of such family. The approach therefore allows modelling a family of SaaS applications and includes a set of variation points to configure them. The resulting feature model configuration can then be used to deploy the application. Similar to our proposal, this approach allows generating the deployment scripts for the application. However, the variation points are too high level features (e.g., “availability”, “environment” or “data separation”) and are not focused on the different features provided by Cloud providers, as in our approach.

While the previous works can provide some interesting ideas to our proposal, their target is not actually the deployment of web applications in Cloud providers. In this sense, there is a shortage of works aiming at Cloud-based solutions and we have only found some approaches created in the context of projects such as Paasage[15] or ModaClouds[16].

Paasage is focused on cloud-based software development and run-time adjustments according to changing execution characteristics. Some of the works developed in the context of this project are related to our proposal. Thus, the work presented in [23] describes a classification of the state-of-the-art of Cloud solutions to help developers to face the heterogeneity among Cloud providers diversity, which hinders the proper exploitation of the full potential of Cloud computing. Additionally, in [24] an architecture to evaluate distributed application deployments in several Cloud providers is presented. The architecture allows developers to better characterize the Cloud needs of their applications and therefore answer important

[14] http://www.webratio.com
[15] www.paasage.eu
[16] www.modaclouds.eu

decisions about which deployment options works best in terms of performance, reliability, cost and combinations thereof. The work presented in [25] describes an approach based on feature modelling and ontologies to handle cloud variability and then manage and create cloud configurations. These works can help within the context of the ARTIST project to better characterize multi-cloud environments as well as some of the requirements to be considered when deploying applications to the Cloud. However, they do not address the problem of generating a deployment for a concrete application to the Cloud.

MODAClouds proposes, similar to ARTIST, a model-based migration approach. However, in MODAClouds the migration of cloud-based software between cloud providers and their interoperability is primarily focused rather than the migration of legacy software to cloud-based software as a means of software modernization. Concerning the works in the context of Cloud deployment, the approach presented in [26] tackles the problem of deploying non-cloud-based applications into the Cloud, however, it does not consider an automatic approach as we propose.

APPENDIX B: Analysis of deployment patterns and frameworks for selected Cloud providers

This section analyses the deployment patterns, techniques and frameworks supported by a set of selected Cloud providers of interest for the ARTIST use cases.

Google App Engine

The Google App Engine supports several different programming languages, i.e., Java, PHP, Python and Go. As a result, the deployment of an application may differ according to the programming language used to implement it. The analysis results presented in the following are restricted to Java-based application.
Google App Engine offers two main procedures supporting the deployment of an application:

- **App Engine Java SDK**, which offers command line tools supporting the deployment.
- **Google Eclipse plugin** which offers Eclipse IDE wizards supporting the deployment.

Deploying a Java-based application requires mainly to upload all the application artefacts, e.g., code bundles, configuration files, libraries, etc., to the Google App Engine. To upload a Java-based application one may either use “appcfg” command provided by the App Engine Java SDK or the Google Eclipse plugin. This plug-in comes basically with a UI for the “appcfg” command.
When using Java for the development, the applications need to be packaged according to the “war-structure” as defined by the JEE specification. In addition to standard JEE configurations, several Google App Engine specific deployment descriptor are available. Some of them are required while others are optional. In this respect, it is important to consider that the Google App Engine offers generally two different kinds of instance types for which specific descriptors need to be provided. So called **Backend Instances** are offered by the Google App Engine mainly to support long-running background processes that are exempted from the 60 second deadline for HTTP requests to **Default Instances**. For a detailed comparison of the two instance kinds, the given table provides a good overview (e.g., backends do not automatically scale, they are billed for uptime rather than CPU usage, etc.).

- **Deployment of applications to Default Instances (Application Configuration)**
  - A Google App Engine Java application must have a descriptor called ***appengine-web.xml*** in its WAR, in the WEB-INF directory. This descriptor,

which refers to the application configuration, needs at least to specify the application ID and its version.
    - Ecore-based meta-model for an overview in github.
- **Backends Configuration**
    - To add backend instances to an application, a descriptor called *backends.xml* is required. At least a name needs to be specified for a backend instance.
    - Ecore-based meta-model for an overview in github.
- **Index Configuration**
    - To go beyond indexes automatically defined by Google App Engine, custom indexes can be specified by a dedicated descriptor called ***datastore-indexes.xml***. An index is basically defined over given properties of entities of a particular kind. The properties are either in ascending or descending order.
    - Ecore-based meta-model for an overview in github.

Amazon WS

Amazon provides services at Infrastructure and Platform level. Usually, Infrastructure and Platform services are accessed via Management Console while SDKs, command-line tools and APIs are provided for access at programming level.

In general there are four ways to interact with Amazon Web Services (AWS):

- Management Console is a graphical user interface enabling user access to Amazon Web Services. Most service features are supported by Management Console, but not all of them.
- AWS Command Line Interface (CLI) is a text-based tool which manages multiple AWS services. There are also other command line tools that enable connection to and communication with AWS services but each of them manages a single service.
- Software Development Kits (SDKs): SDKs provide a way to programmatically access Amazon's services simply by adding class libraries to the application's code and using them in order to communicate with the desired service features.
- Low-Level APIs: Query, REST and SOAP APIs. Query and REST APIs use the standard components of HTTP request messages but in a different way. Their difference is that REST APIs use the HTTP methods in order to describe the action to be performed while in Query APIs the action is described through parameter values (together with the data the action will be performed on). On the other hand, SOAP APIs use SOAP xml documents constructed as another layer on top of the HTTP protocol.

Deploying an application

There are four possible ways to deploy an application. Starting from Elastic Beanstalk (see Figure 31 below) and [descending] following the direction to the right, the user gains control of the deployment and execution management but gives up in automation. Access to these deployment tools is given via Management Console and SDKs or CLI.

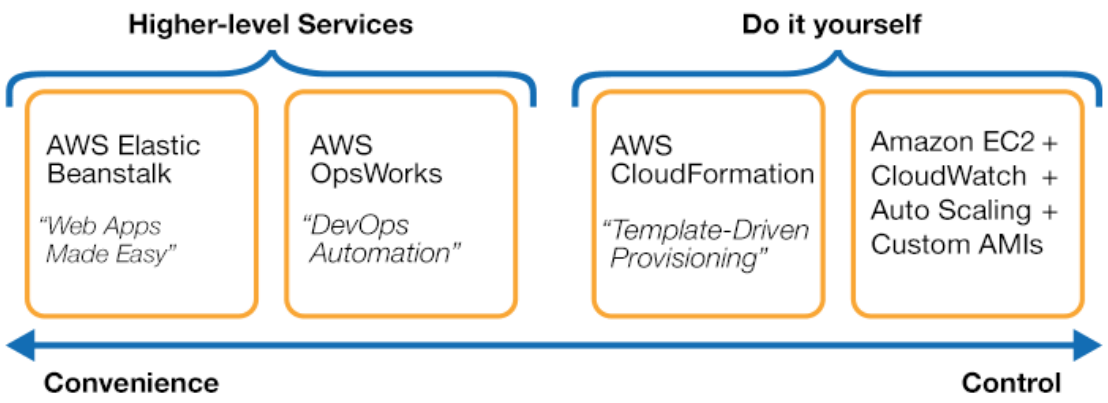

Figure 31 AWS Deployment Services

Elastic Beanstalk is a service used for development, testing and deployment. In terms of deployment, Elastic Beanstalk provides a PaaS-like deployment solution as the user doesn't have to know anything about AWS infrastructure in order to deploy the application. Having built the application, deploying it becomes very easy, simply by downloading the application package (zip or war files), choosing names and urls, and selecting between two types of environments (loadbalanced and single VM environments). For Java application, Java AWS SDK can be used in order for the application to be developed, built and run locally in the Eclipse IDE, before the Elastic Beanstalk deployment. In the same way, .NET SDK can be used together with AWS toolkit for VisualStudio in order to develop and run application locally. User can also make environment configurations using configuration templates, Must be noted that, in order to use an RDS (Relational Database System) instance together with the deployed application, the instance must be created and configured before deployment (can also be done through Elastic Beanstalk). This service also enables updating a deployed application. Application's versions and environment configurations are stored in S3 (Simple Storage Service).
Restrictions: Elastic Beanstalk supports specific container types (Infrastructure topology and software stack to be used for a specific environment). For Java applications the supported container types include Amazon Linux running Apache Tomcat, whereas for .NET applications Windows Server running IIS 8 or IIS 7.5.

OpsWorks is a service designed to enforce management through the whole application's lifecycle. The main features of OpsWorks are:

1. **Stacks.** A stack is a set of instances that user wants to be managed collectively because they serve the same purpose (e.g. serve all functionalities of an application).
2. **Layers.** A layer in a stack expresses the functionality of instances included in this layer and defines all the packages, the applications and the configurations that are essential for these instances. For example an instance belonging to a loadbalancing layer should be able to distribute incoming traffic to application servers.
3. **Instances.** An instance is an EC2 (Elastic Compute Cloud) instance determined to serve the above mentioned functionalities defined by a stack layer.

The whole process of getting an application running using OpsWorks is completed in the according steps:

1. Creation of a stack.
2. Definition of stack layers. There are some pre-built layers that support standard application frameworks. However, user can define his own stack layers. In addition, as far as working with databases is concerned, while pre-built layers include only MySQL, user can install any other database on EC2 instances using custom layers or chef

recipes (e.g. Cassandra, PostgreSQL). In addition, Chef recipes can be used in order to establish a connection to an existing RDS instance or DynamoDB table. Application layers:

a. Ruby on Rails, PHP, Node.js, Java and Nginx
b. Data Layers: MySQL and Memcached
c. Utility Layers: Ganglia and HAProxy

3. Assign instances to the layers. This step is about creating the instances with the chosen configurations.
4. Application deployment. In this step user must specify where the code is placed (supported repositories: Git, SVN, HTTP and S3), and any additional deployment tasks such as database configuration.

It must be noted that OpsWorks uses chef integration framework in order to automate the deployment of applications. More specifically, OpsWorks uses OpsCode Chef cookbooks for all the deployment installation and configuration of tasks, such as scripts execution. As mentioned at the second step of application deployment, there are pre-built cookbooks, but in order to define a specific stack layer, these cookbooks can be extended or overwritten through the implementation of custom cookbooks. A cookbook consists of:

1. Attributes files (files containing attributes to be used by recipes and templates).
2. Template files (templates that recipes use to create other files such as configuration files).
3. Recipes (Ruby applications that define every task needed to configure a system).

Recipes can be executed automatically by being attached at a layers lifecycle or manually by running the corresponding CLI command.

Restrictions: At the moment OpsWorks supports only Amazon Linux and Ubuntu 12.04 LTS among custom AWS AMIs (Amazon Machine Images).

CloudFormation is a provisioning and deployment service based on JSON formatted text files called templates. Templates are used to describe the AWS infrastructure needed for the execution of an application as well as the inter-connection between them. When the JSON file is created, it is used as the base for a stack creation. A stack is the set of all the initiated resources.

The top level JSON Objects contained in the template files are:

1. Description: A text description for the template usage.
2. Parameters: A set of inputs used to customize the template per deployment.
3. Resources: The set of AWS resources needed and the relationships between them.
4. Outputs: A set of values to be made visible to the stack creator.
5. AWSTemplateFormatVersion: Date of the tool version to be used (if not created, the latest version is assumed)

CloudFormation provides an IaaS deployment solution helping mostly with resource provisioning. However, the management of resources and the execution of the application is still under users' control.

One of CloudFormation's interesting features is the allowance of scripts to be executed at the initial boot of the instantiated resource. In addition, some helping scripts are available, which, among other functions, automate the essential download and installation of files and packages, as well as with signalling the stack creation workflow that the application is up and running. So, the main steps for deploying an application in an existing or an under construction stack are to get application package onto a downloadable location, include userdata in the CloudFormation JSON file and execute them using the corresponding helper script, provide location to download source files and zips in the metadata section.

There are some differences between deployment in linux and Windows based AMIs but the main concept remains the same.
(Note: CloudFormation is also accessible via AWS Toolkit for Visual Studio and Eclipse as a user friendly solution)

Manual deployment by infrastructure resource accessing and managing (direct use of services)

Amazon EC2 instances are the virtual machine instances that form the fundamental compute block of a deployment environment. Instances are created from AMIs which contain a pre-defined operating environment. User has also the choice of creating and uploading his own AMI. There are many EC2 instance types, according to user's needs for size, computing power etc. Except for APIs and CLI, all instance configurations can be made through the supporting Management Console.
In order to use Amazon EC2 service, one must take under consideration the following features:

1. Key pairs
2. Temporary and persistent storage volumes
3. Physical locations for resources (regions and Availability zones)
4. Security groups that function as a firewall
5. Elastic IP addresses
6. tags (metadata assigned to EC2 resources)

So, using this strategy, every infrastructure component used for the deployment (compute, storage, networking, load balancing, etc.) must be configured manually.

Next table in Figure 32 describes all the possible ways for accessing the majority of Amazon Web Services, including the most important ones in terms of infrastructure provisioning, management and automating the application deployment.

Compute	Query API	REST API	JAVA SDK	.NET SDK	CLI	PHP SDK	Python SDK	Windows Powershell	Android SDK	JavaScipt (Browser)	Node.js
EC2	YES		YES	YES	YES	YES	YES	YES	YES		YES
Auto Scaling	YES		YES	YES	YES	YES	YES	YES	YES		YES
Elastic Load Balancing	YES		YES	YES	YES	YES	YES	YES	YES		YES
WorkSpaces											
Networking											
Virtual Private Cloud	YES		YES	YES	YES		YES				
Route 53		YES		YES	YES	YES	YES	YES			YES
DirectConnect	YES			YES	YES	YES	YES	YES			YES
Storage											
Glacier		YES		YES		YES	YES				YES
Elastic Block Storage	YES										
S3		YES	YES	YES	YES	YES	YES	YES	YES	YES	YES
Import/Export	YES		YES	YES	YES	YES	YES	YES			
Storage Gateway	YES			YES	YES	YES	YES	YES			YES
CDN											
CloudFront		YES	YES	YES		YES	YES	YES			YES
Database											
ElastiCache	YES			YES	YES	YES	YES	YES			YES
Relational Database Service	YES		YES	YES	YES	YES	YES	YES			YES
RedShift (warehouse)	YES			YES	YES	YES	YES	YES			YES
DynamoDB	YES			YES	YES	YES	YES	YES	YES	YES	YES
SimpleDB	YES		YES	YES		YES	YES		YES		YES
Data Processing											
Elastic MapReduce	YES		YES	YES		YES	YES	YES			YES
Kinesis	YES					YES		YES			
DataPipeline	YES			YES		YES	YES	YES			YES
Application Services											
AppStream		YES									
CloudSearch		YES		YES		YES	YES	YES			YES
Simple Queue Service	YES		YES	YES	YES	YES	YES	YES	YES	YES	YES
Simple Email Service	YES		YES	YES	YES	YES	YES	YES	YES		YES
Simple Notification Service	YES		YES	YES	YES	YES	YES	YES	YES	YES	YES
Simple Workflow Service	YES			YES	YES	YES	YES				YES
Flexible Payments Service											
Elastic Transcoder	YES			YES	YES	YES	YES	YES			YES
Deployment and Management And monitoring											
CloudWatch	YES		YES	YES	YES	YES	YES	YES	YES		YES
Elastic Beanstalk	YES		YES	YES	YES	YES	YES	YES			YES
CloudTrail	YES					YES		YES			
Identity and AccessManagement	YES		YES	YES	YES	YES	YES	YES			YES
Management Console											
CloudFormation	YES		YES	YES	YES		YES	YES			YES
OpsWorks	YES				YES	YES	YES	YES			YES
CloudHSM											YES
Security Token Service			YES			YES	YES	YES	YES	YES	YES

Figure 32 Compatible clients for AWS Deployment Services

Microsoft Azure

There are three ways to access a Windows Azure environment to perform management tasks, such as deploying and removing roles and managing other services:

- The first is through the **Windows Azure Management Portal**, where a single Microsoft account has access to everything in the portal.
- The second is by using the **Windows Azure Service Management API**, where API certificates are used to access to all the functionality exposed by the API.
- The third is to use the **Windows Azure Management PowerShell** cmdlets.

The Windows Azure PowerShell cmdlets use the Windows Azure Service Management REST API to communicate with Windows Azure. The communication is secured with a management certificate, which is downloaded and installed on the client machine as part of the Windows Azure PowerShell cmdlets installation. This means you are not prompted for credentials when you use these cmdlets.

There is a fourth way, which is implementing your own deployment via code. A number of SDK's have been created, using these same Management API's, for different languages (PHP, Java, Python, Ruby, Node.js, etc.).

Subscription
The Windows Azure Service Management API uses mutual authentication of management certificates over SSL to ensure that a request made to the service is secure. No anonymous requests are allowed. The subscription (with a subscription-id and an associated certificate) is a unique user account on Windows Azure and is a very important concept since you cannot execute any task without it.

A **Publish Settings** file is an XML file which contains information about your subscription. It contains information about all subscriptions associated with a user's Live Id (i.e. all subscriptions for which a user is either an administrator or a co-administrator) as well as the certificates. This allows easing deployment through Visual Studio or other tools that are able to work with these.

IaaS

The preferred way to deploy your infrastructure is through PowerShell. There are two kinds of files relevant here:

- Deployment scripts. Scripts like these setup the deployment. The main concepts being created and/or configured are:
  - **Affinity Group**: This is a logical construct associated with a geo-region and defined at the subscription level.
  - **Availability Sets**: An Availability Set is a logical group to signify the need for Windows Azure to prevent a single point of failure for all VMs included in the set.
  - **Storage Account**: A storage account provides the access to Windows Azure storage within a geographic region. There are three types of storage: Blob, Queue, and Table in Windows Azure.
  - **Cloud Service**: This is a logical container including application code and configurations. For Windows Azure IaaS, each VM is deployed to a service, however a service can contain multiple VMs. Placing multiple VMs into a service makes these VMs connected and visible to one another.
  - **Virtual Network**: Define settings of the Virtual Network (typically through a dedicated file).
  - **Virtual Machine**
  - **Load Balancer**
- The **Network Configuration** file, which describes Virtual Network configuration settings. The default extension is .netcfg.
  - **VirtualNetworkConfiguration** specifies Virtual Network and DNS values

Sidenote: You can also further automate the deployment and removal of additional instances based on demand using a framework such as the Enterprise Library Autoscaling Application Block. This allows to set so-called rules for dynamically changing the configuration of your system:

- *Constraint rules enable you to set minimum and maximum values for the number of instances of a role or set of roles based on a timetable.*
- *Reactive rules allow you to adjust the number of instances of a target based on aggregate values derived from data points collected from your Windows Azure environment or application.*

PaaS

To deploy an application to Windows Azure Cloud Services three files are involved:

- The **Service Package** file that contains all your application's files which can be generated using the Cspack.exe command-line utility or through Visual Studio.
- The **Service Definition** file describes the service model. It defines the roles included with the service and their endpoints, and declares configuration settings for each role. The default extension for the service definition file is .csdef. The main concepts defined here are those of:
  - **Web Role** (Web Application Programming)
  - **Worker Role** (Background processing)
- The **Service Configuration** file specifies the number of instances to deploy for each role and provides values for any configuration settings declared in the service definition file. The default extension for the service configuration file is .cscfg. The most important concepts here are:
  - **Role** (specifies the number of role instances to deploy for each role in the service, the values of any configuration settings, and the thumbprints for any certificates associated with a role).
  - **NetworkConfiguration** (deployment of cloud services in Virtual Networks, also important here are configurations related to access control).

Typically, multiple Service Configuration files are used. For example, one for testing locally on the Azure emulator, and a number of additional ones for deploying onto the Azure cloud subscription (either test/staging/production environments).

In addition, also a number of configuration settings, such as connection strings and authentication information for the application, are still stored using the **Web.config** file. This follows the same paradigm as before with the on-premise versions of the web applications.

Since it is not easy to edit the Web.config file when an application is deployed to Windows Azure Cloud Services (i.e. you must redeploy the entire application when values need to be changed), a lot of the times some application configuration settings are moved from the Web.config file to the service configuration file. However, this gives problems since some components cannot read settings from this service configuration file. Therefore, typically a number of scripts are built to make those configuration changes on the fly.

You can deploy an application by uploading the files using the Windows Azure Management Portal, by using the Publish Windows Azure Application wizard in Visual Studio, or the automated way by using Windows Azure PowerShell cmdlets. Both the Visual Studio wizard and the PowerShell cmdlets authenticate with your subscription by using a management certificate instead of a Microsoft account. The automated deployment of an application in production is in most cases handled in multiple stages. The first stage uses an MSBuild script to compile and package the application for deployment to Windows Azure. This build script uses a custom MSBuild task to edit the configuration files for a cloud deployment, adding the production storage connection details. The second stage uses a Windows PowerShell script with some custom cmdlets to perform the deployment to Windows Azure.

APPENDIX C: Analysis of platform-independent deployment patterns and entities

From the platform dependent analysis produced in the previous section, this one proposes a platform independent description of the deployment patterns and entities.

Platform independent meta-model for deployment patterns

This section elaborates a platform independent meta-model that supports the specification of deployment patterns and requirements. These concepts will contribute to the extension of CloudML@ARTIST to support deployment needs.
Deployment platform independent concepts and entities have been derived from the analysis of the deployment patterns and frameworks supported by the target Cloud providers, conducted in APPENDIX B. From this analysis, we have provided two meta-models, one from the provider perspective (see Figure 33) and another one from the application owner perspective (see Figure 34), which contain deployment concepts, entities and their relationships among both meta-models. Technically these meta-models have been implemented as instances of the EMF Ecore meta-meta-model[17]. Both deployment PI meta-models are located in the ARTIST-Tooling Github repository at:
https://github.com/artist-project/ARTIST-Tooling/tree/master/migration/modernization/eu.artist.migration.modernisation.dt.model/model/PIMeta-modelv0.3

The color schema used for entities in these meta-models has the following meaning:

- Pink colored concepts are those already included in CloudML@ARTIST
- Blue colored concepts are those Cloud provider related concepts NOT included in CloudML@ARTIST
- Yellow colored concepts are those application owner related concepts NOT included in CloudML@ARTIST

[17] http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html#details

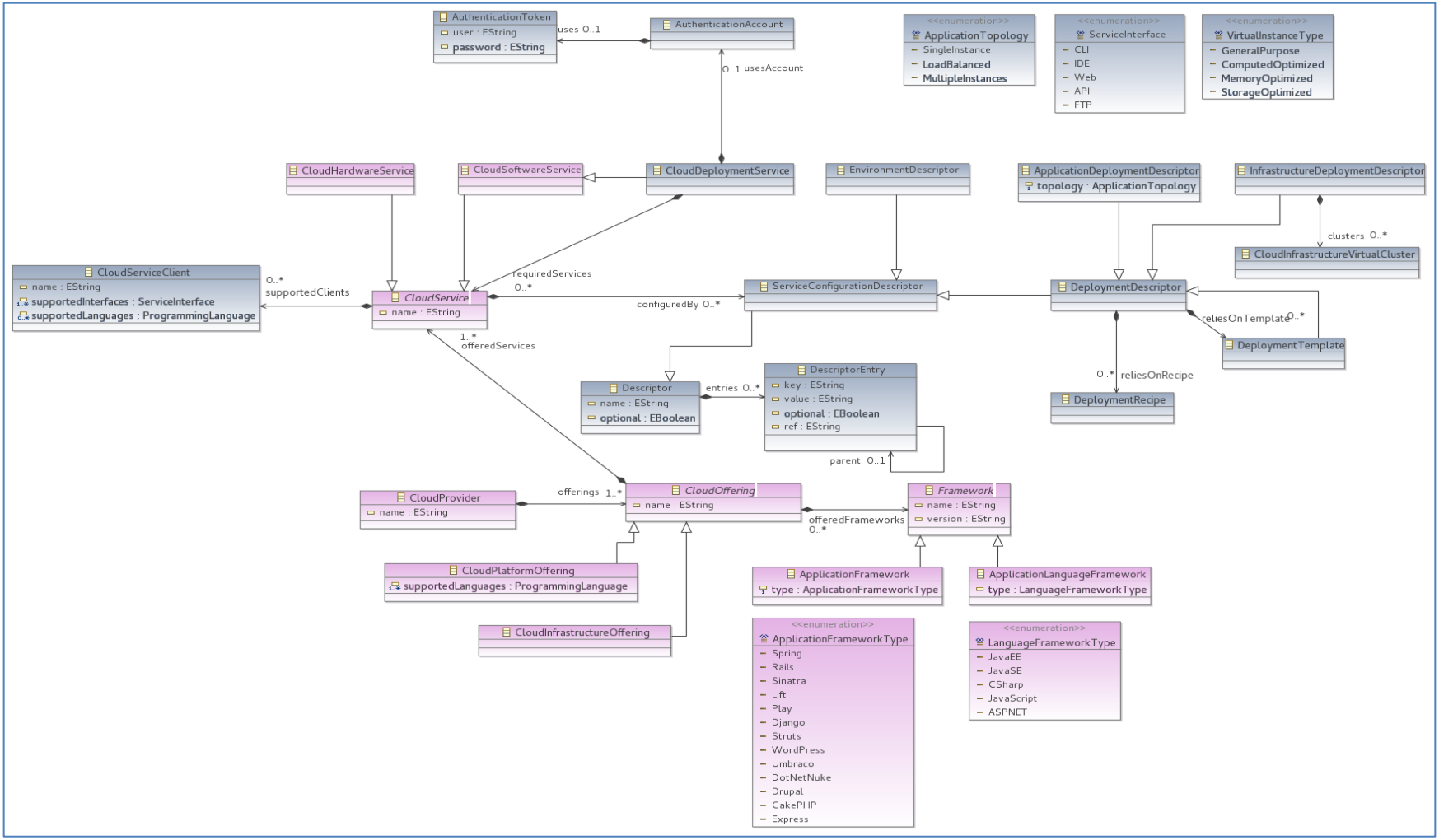

Figure 33 Deployment platform-independent meta-model for Cloud provider perspective

The PI meta-model for Cloud-provider-perspective support for deployment describes all those deployment concepts and entities concerning the Cloud provider (see Figure 33). A CloudProvider offers several CloudOfferings, whose possible types are CloudPlatformOffering (e.g. PaaS) and CloudInfrastructureOffering (e.g. IaaS). Although typically the relation between CloudProvider and CloudOffering is 1:1, we support the provider can offer more than one offering. A CloudOffering is a collection of offered CloudServices and software Frameworks. Framework types are:

- ApplicationFrameworks: typically required third party frameworks, including not only runtime libraries but other runtime applications required by the application to be deployed (i.e. Spring, Wordpress).
- ApplicationLanguageFrameworks: baseline runtime frameworks supporting the execution of the deployed application specific to the language the application was implemented with (i.e. J2SE, .NET)

The CloudServices are those Cloud provider specific individual services that constitute the offering portfolio. These CloudServices could be of type:

- CloudHardwareServices are virtualised services offering hardware utilities (e.g. IaaS), such as computing, storage, network or memory.
- CloudSoftwareServices are platform services offering software capabilities to deployed applications, such as security, persistence, application containers, etc.

For each CloudService in its offering, the CloudProvider typically offers one or more CloudServiceClients to its customers, as facilities to be installed and used on the customer side (i.e. local computer), enabling one or more ServiceInterfaces (i.e. API, CLI, IDE, Web, etc).
Additionally, the customer usage of each CloudService is optionally configured by zero or more ServiceConfigurationDescriptors. Any Descriptor includes a set of DescriptorEntries (characterised by a unique key) which can reference their parent in order to support tree-based nested configuration structures. A DescriptorEntry also contains a reference to the CloudML@ARTIST stereotype property that references this entry value in an application model properly annotated with deployment requirements.
A particular kind of CloudSoftwareService is the CloudDeploymentService, which supports the deployment of applications into the CloudOffering. This service is configured by one or several DeploymentDescriptors. Different kinds of DeploymentDescriptors have been identified:

- ApplicationDeploymentDescriptor: typically describes the deployment configuration for an application into a CloudPlatformOffering
- InfrastructureDeploymentDescriptor: typically describes the deployment configuration for a virtual image into a CloudInfrastructureOffering. This descriptor enables the configuration of a set of CloudInfrastructureVirtualImages, optionally grouped into CloudInfrastructureVirtualClusters.
- DeploymentTemplate: reusable pre-configured DeploymentDescriptor for some typical usages that can be further customised by the user.

A DeploymentDescriptor can reuse optionally predefined and reusable DeploymentRecipes, which automates the installation of predefined frameworks, typically into CloudInfrastructureVirtualImages.
Another kind of ServiceConfigurationDescriptor is the EnvironmentDescriptor, which enables the specification of the environment within the application is running, through the setting of environment variables.

Finally this meta-model also includes a set of enumerations that provide concrete instances for some entities referenced in the meta-model.

The PI meta-model - application-owner-perspective support for deployment - describes all those deployment concepts and entities concerning the application owner (Figure 34).
A CloudApplication comprises a set of ApplicationComponents (kind or CloudApplications as well). This distinction is only required to support composability on deployment. Either the application itself or the individual components can be deployed separately, depending on the owner needs. Each CloudApplication can be configured by some ApplicationDescriptors (subtypes of Descriptor).

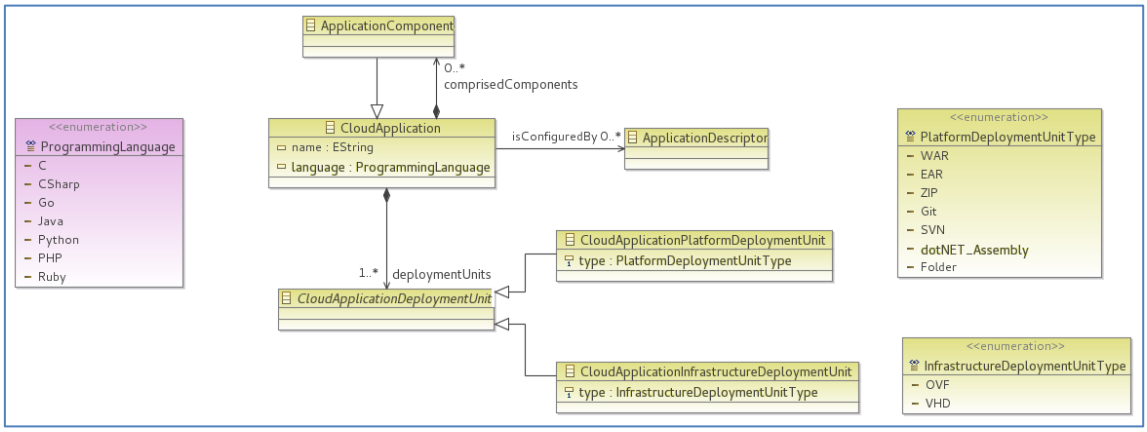

Figure 34 Deployment platform-independent meta-model for application perspective

Those descriptors (i.e. web.xml for J2EE applications) characterise the application itself but not the deployment configuration (e.g. ApplicationDeploymentDescriptor such as the Google App Engine application.xml). In order to support its deployment, each CloudApplication can be bundle into one or more CloudApplicationDeploymentUnits, whose types could be either a CloudApplicationPlatformDeploymentUnit (e.g. intended for deployment into a platform, PaaS) or a CloudApplicationInfrastructureDeploymentUnit (e.g. intended for deployment into an infrastructure, IaaS). Examples of platform and infrastructure deployment unit types are given in corresponding enumerations.

The existing relations between concepts of both meta-models are not explicitly renderer in these above figures, due to functional limitations on EMF framework, but are they explicitly included in both meta-models and commenting in the following (see Figure 35).

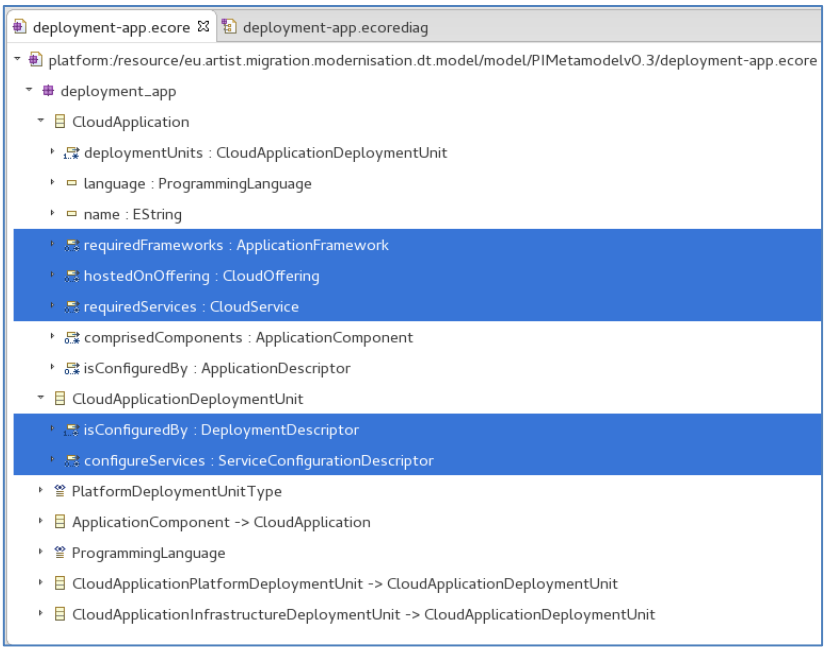

Figure 35 Cross-referencing meta-model concepts for deployment

As shown in Figure 35, the deployment platform-independent meta-model for application perspective imports and references concepts defined in the meta-model for Cloud provider perspective. In particular, a CloudApplication may require third party ApplicationFrameworks and CloudServices offered by the CloudOffering, where will be hosted. The CloudApplicationDeploymentUnit is configured by DeploymentDescriptors and also configures required CloudServices through ServiceConfigurationDescriptors. These CloudApplication and CloudApplicationDeploymentUnit properties are referencing these mentioned concepts, which are defined in the other meta-model.

Platform Domain Models for Cloud providers

This section elaborates the concrete PDM instances (conforming to the deployment PI meta-model) corresponding to the selected target Cloud providers: Google App Engine, Amazon Web Services and Microsoft Azure.

Google App Engine PDM

An initial version of the Google App Engine deployment PDM instance is located in the ARTIST-Tooling Github repository at:
https://github.com/artist-project/ARTIST-Tooling/tree/master/migration/modernization/eu.artist.migration.modernisation.dt.model/model/PIMeta-modelv0.3/CloudOfferingPDM/

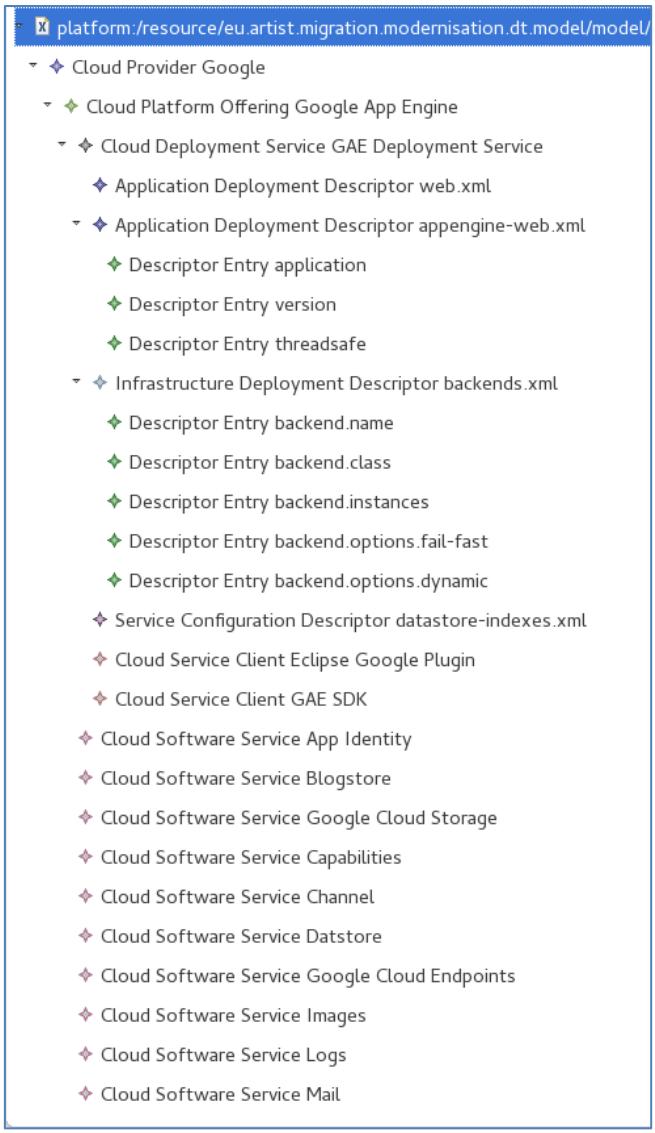

Figure 36 Google App Engine PDM snippet

Amazon Web Service PDM

An initial version of the Amazon Web Service deployment PDM instance is located in the ARTIST-Tooling Github repository at:
https://github.com/artist-project/ARTIST-Tooling/tree/master/migration/modernization/eu.artist.migration.modernisation.dt.model/model/PIMeta-modelv0.3/CloudOfferingPDM/

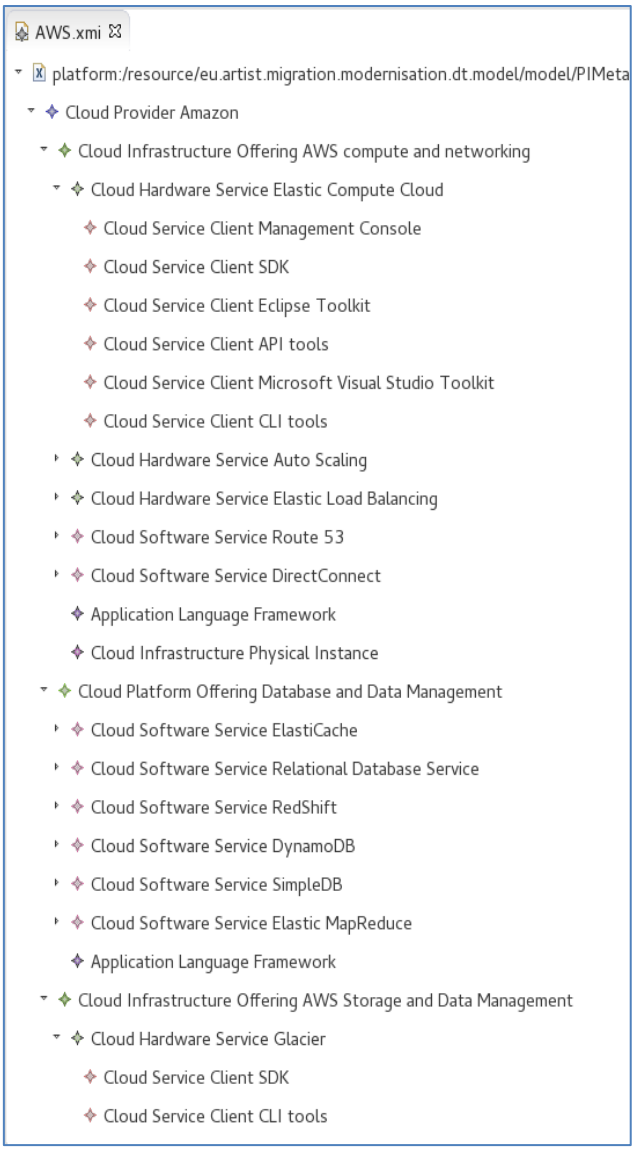

Figure 37 Amazon Web Service PDM snippet

This is the model describing the Amazon Cloud Provider. Amazon provides the AWS composing, thus, a set of offerings most of which are categorized as Cloud Infrastructure Offerings (not all of them, though). At the moment, four major Cloud Offerings are included:

- "Compute and networking" offers the main AWS service, EC2 which provides the means for configuring and controlling the computing resources, as well as a set of other related services working with EC2 while covering scalability and networking aspects.
- "Database and Data Management" and "Storage and Data Management" offer data related services at platform and infrastructure level respectively.
- "Deployment and Management" offers the three services which facilitate and, in some level, automate the deployment on AWS infrastructure (CloudFormation, OpsWorks and Elastic Beanstalk) as well as CloudWatch, which enables monitoring and thus controlling the deployed AWS resources.

AWS offers a variety of choices for deploying an application, the easiest of which is the Elastic Beanstalk deployment, imposing however a set of constraints. Either using one of the deployment services or the AWS infrastructure directly (EC2, S3, Elastic LoadBalancing, Auto Scaling), there is a number of service clients which enable communication with AWS: SDKs (including Java, .Net, Python, Ruby and PHP), CLI tools and the Management Console (web-based user interface). In addition Eclipse and Visual Studio toolkits have been developed which facilitate the development as well as the deployment by permitting communication with some AWS resources, incorporating .NET and Java SDKs and providing support for some deployment services.

Azure PDM

An initial version of the Microsoft Azure deployment PDM instance is located in the ARTIST-Tooling Github repository at:
https://github.com/artist-project/ARTIST-Tooling/tree/master/migration/modernization/eu.artist.migration.modernisation.dt.model/model/PIMeta-modelv0.3/CloudOfferingPDM/

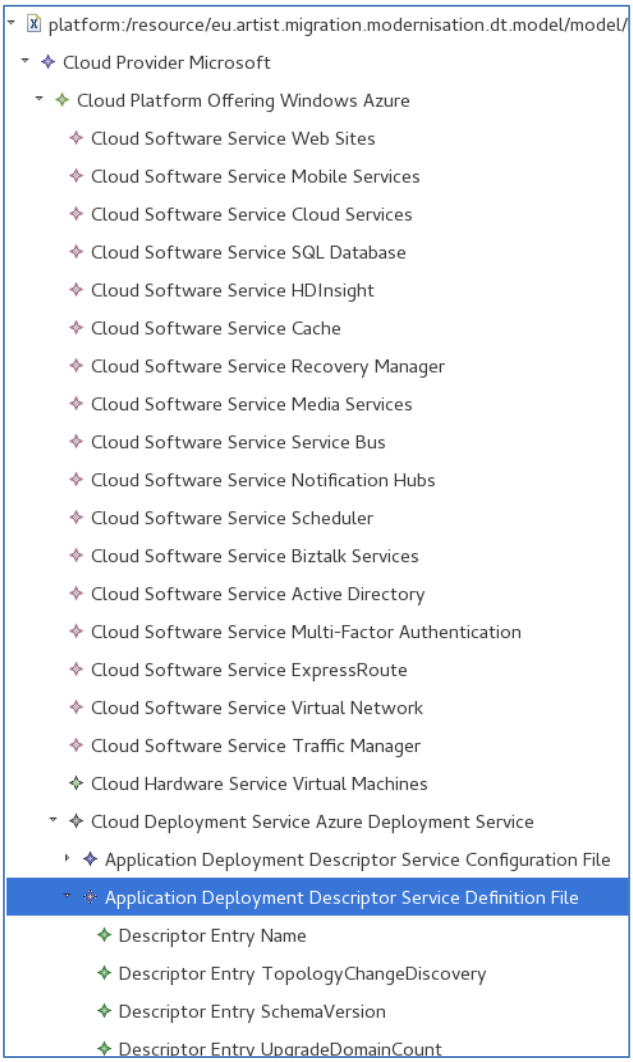

Figure 38 Microsoft Azure PDM snippet

This model describes Microsoft as the Cloud Provider. There are basically two Cloud Offerings defined: Windows Azure and Windows Azure Pack. The former relates to the public cloud offering while the latter refers to the on-premise private cloud offering.

The main offering of Microsoft is Windows Azure. This offering contains about 18 software services: Web Sites, Mobile Services, Cloud Services, SQL Database, HDInsight, Cache, Recovery Manager, Media Services, Service Bus, Notification Hubs, Scheduler, Biztalk Services, Active Directory, MFA, ExpressRoute, Virtual Network and Traffic Manager. Furthermore it contains three services categorized as hardware services: Virtual Machines, Storage and Backup. In Azure, there are many ways/clients to deploy your application. These are defined in the Deployment Service. One can use the well-known developer tools (IDEs) such as Visual Studio and/or WebMatrix to deploy and/or upload (via FTP) the application. For Java-based applications, there is also a plugin created for the Eclipse environment. Most of the times, the easiest way to deploy is via the Management Portal. However, automating deployment would probably benefit most from the SDK (available in CSharp, Ruby, Python, Java) and the Management tools that are provided either as a set of Powershell commandlets or a true CLI. Deploying an application typically involves two Application Deployment Descriptors, the Service Definition File and the Service Configuration File, and one (optional) Infrastructure Deployment Descriptor, the Network Configuration File. In simple applications these files do not contain much (only basic information such as the Name, Role Name, Number of Instances, Size of VM and so on) and are usually automatically generated by the developer tools. All the clients are internally using the publicly available Management API (JSON/REST Service).

The Azure Pack offers a limited set of Azure technologies / services for your own data center. These services are Web Sites, Service Bus, Virtual Machines, Virtual Network and SQL Database. All these services are consistent with their Azure (public cloud) counterparts. The deployment service consists of 2 clients being put forward here, the Service Management API and the Management Portal[18].

[18] Although not explicitly stated in documentation, also the other deployment mechanisms such as via the development environments can also be used.