BPAAS ALLOCATION AND EXECUTION ENVIRONMENT PROTOTYPES

D3.4

Editor Name	Frank Griesinger (UULM)
Submission Date	December 31, 2016
Version	1.0
State	FINAL
Confidentially Level	PU

Co-funded by the Horizon 2020
Framework Programme of the European Union

EXECUTIVE SUMMARY

The output of this deliverable are the research prototypes that were developed to be fed back to the stable branch of the Execution Environment and Allocation Environment, as soon as they reach the needed maturity. The development is based on a selection of the discussed and presented blueprints in Deliverable D3.3. The prototypes deal with interesting research problem that appear in the context of BPaaS management. They are covering the modelling of BPaaS bundles, the discovery and supportive mapping of services, the component orchestration at the PaaS level and the synergic cross-layer BPaaS monitoring and adaptation.

This report: (1) provides a problem statement and a running example to exemplify the needs and solutions in the research related to the BPaaS Execution and Allocation Environments, (2) explains the BPaaS modelling approaches towards multi-level cloud support and BPaaS adaptation rule specification, (3) analyses the (BPaaS) allocation research towards smart service discovery and composition as well as a supportive deployment modelling based on DMN, and (4) explicates the run-time management support over BPaaS with respect to multi-level deployment and provisioning as well as cross-layer monitoring and adaptation.

The CAMEL cloud-domain language was enhanced to surpass the state-of-the-art and satisfy the requirements from the use cases of the project. To this end, it was extended with the capability to model PaaS services and to include them in the description of cross-layer (BPaaS / application) deployment plans as well as specify advanced adaptation rules with sophisticated composite adaptation plans / strategies. Furthermore, an extensive SLA support in terms of an OWL-Q extension has been realised.

The Smart Service Discovery and Composition prototype enables precisely and semantically discovering services based on both the functional and non-functional aspects as well as to compose them according to global non-functional user requirements (e.g., cost, QoS, and security). For service composition, a sophisticated service selection algorithm has been proposed which is able to cover simultaneously both the IaaS and SaaS level. The DMN-based CAMEL description approach aims at supporting the definition of mappings between the several levels of the BPaaS life cycle, such as business plan, workflow or executable workflow.

The aforementioned CAMEL extensions are currently being realised in the prototypes for PaaS orchestration and cross-layer monitoring and adaptation and will be reflected in the Cloud Provider Engine and multiple other components of the Execution Environment, including the Monitoring and Adaptation Engine.

The stable as well as the research version of the prototypes are partly available free for download from the CloudSocket webpage (cloudsocket.eu/download). The Cloudiator application is constantly merged with the stable branch to provide best possible stability but newly integrated features are in an experimental state. The other described prototypes will be released under the specified license of each partner.

PROJECT CONTEXT

Workpackage	WP3: Business Process as a Service Research
Task	T3.2: BPaaS Allocation and Execution Environment Research
Dependencies	Input to D3.5, T3.3 and WP4

Contributors and Reviewers

Contributors	**Reviewers**
Frank Griesinger, Daniel Seybold, Jörg Domaschka (UULM), Kyriakos Kritikos (FORTH), Chrysostomos Zeginis (FORTH), Román Sosa Gonzalez (ATOS)	Andreea Popovic (YMENS) Knut Hinkelmann (FHNW) Antonio Gallo (FHOSTER)

Approved by: Stefan Wesner (UULM) as WP 3 Leader

Version History

Version	**Date**	**Authors**	**Sections Affected**
0.1	November 22, 2016	Daniel Seybold (UULM)	Initial version, TOC
0.2	November 29, 2016	Frank Griesinger (UULM)	All
0.3	December 05, 2016	Roman Sosa (ATOS)	PUL
0.4	December 07, 2016	Kyriakos Kritikos (FORTH)	All
0.5	December 08, 2016	Daniel Seybold (UULM)	DMN
0.6	December 11, 2016	Frank Griesinger (UULM)	All
1.0	December 15, 2016	Frank Griesinger, Daniel Seybold (UULM)	All, cleaning

TABLE OF CONTENT

LIST OF FIGURES

1 INTRODUCTION AND PROBLEM STATEMENT

This deliverable introduces the research contribution for the Allocation and Execution Environment in form of selected prototype implementations of the Allocation and Execution Environment Blueprints of D3.3 [1]. As highlighted in D3.3 the blueprints are categorized in BPaaS Modelling, Allocation Environment and Execution Environment blueprints and hence, the analysed prototypes of this deliverable follow the same structure.

1.1 Project Context and Blueprint selection

The BPaaS Allocation and Execution Environment prototypes present the technical realisation of the respective Blueprints of deliverable D3.3 in order to showcase their technical feasibility and the actual enhancement to the CloudSocket platform. As described in deliverable D3.3, each of the three blueprint categories contains multiple research assets, though not all assets will be implemented as prototypes. Therefore, in cooperation with WP4, a selection of prototypes was agreed on for each blueprint category. The selection was driven by novelty from the WP3 perspective and also by functional benefits from WP4 perspective.

The selected prototypes focus on extension of the BPaaS deployment and orchestration capabilities by including the PaaS level within the *PaaS Orchestration* prototype in the BPaaS Execution Environment. This extension also affects the BPaaS Modelling on the CAMEL level and the BPaaS bundle creation in the BPaaS Allocation and Execution Environments.

The BPaaS Allocation Environment prototypes target the ease of creating the technical BPaaS bundle specification by applying the prototypes of *Smart Service Discovery and Composition* and *DMN-to-CAMEL Mapping*.

Additional BPaaS Execution Environment prototypes target the holistic monitoring and adaptation across all cloud service levels (from the infrastructure to the workflow level) by applying the *Synergic Cross-Layer Monitoring Framework* and the *Synergic Cross-Layer Adaptation Framework*.

Figure 1 shows the high-level architecture of the CloudSocket platform and highlights the components, which are enhanced with research prototypes. The yellow boxes refer to the BPaaS Modelling prototypes, the red boxes to the Allocation Environment prototypes and the green boxes the Execution Environment prototypes.

As this deliverable focuses on the technical realisation of the selected prototypes, for each prototype a brief feature description with its integration into the existing environment and the benefits for the CloudSocket platform is provided.

In order to ease the technical details for each prototype, an architectural overview is provided, containing the internal components and the exploited interfaces to existing environment components. This also includes a setup guide, explaining all technical requirements and configuration steps.

In addition, for each prototype a future work section provides an overview of possible enhancements in the scope of WP4, if the prototype is going to be integrated into the production environment[1].

[1] The „production environment" is a stable instantiation of the CloudSocket tool-suite, that remains in a stable state. These will further be elaborated in T4.5.

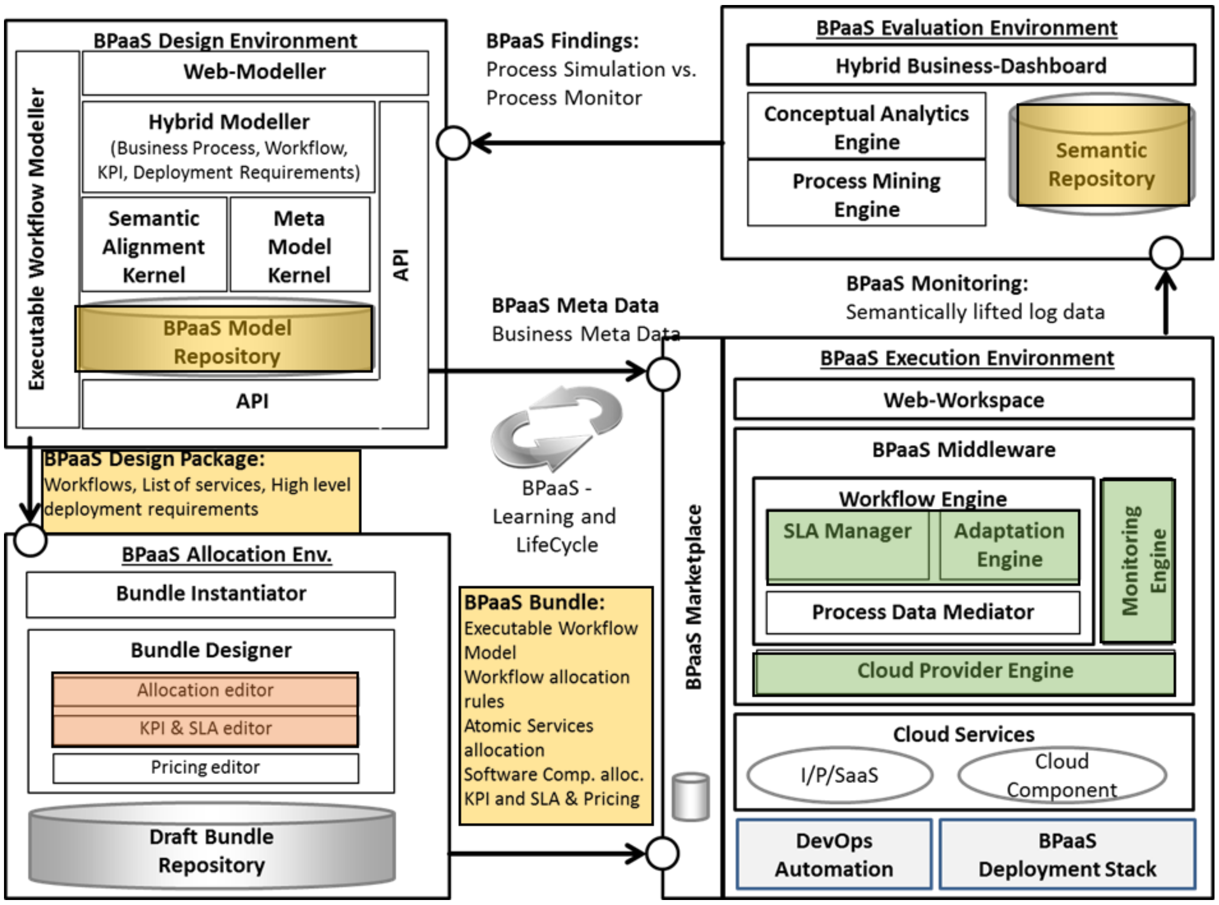


Figure 1 Context to the architecture of the CloudSocket platform

The presented blueprints are also correlated to those presented in D3.5 [7]. This particularly applies for the Adaptation, Monitoring and Cloud Provider Engines from which respective information is harvested for the analysis purposes in [7]. More details about the respective blueprints for the BPaaS Evaluation Environment research prototype can be found in [7], while specific information about the input required for the latter research prototypes proper functioning can also be found in D3.3 [1].

1.2 Running example

In order to ease the understanding of the research prototypes over the current CloudSocket solutions, we use the ChristmasCardDesigner service of the “Sending Christmas Greetings” business process [2] as a running example to demonstrate the respective prototype features. The complete business process of “Sending Christmas Greetings” is shown in Figure 2, where the activities of the Christmas Card Designer are shown in the top lane.

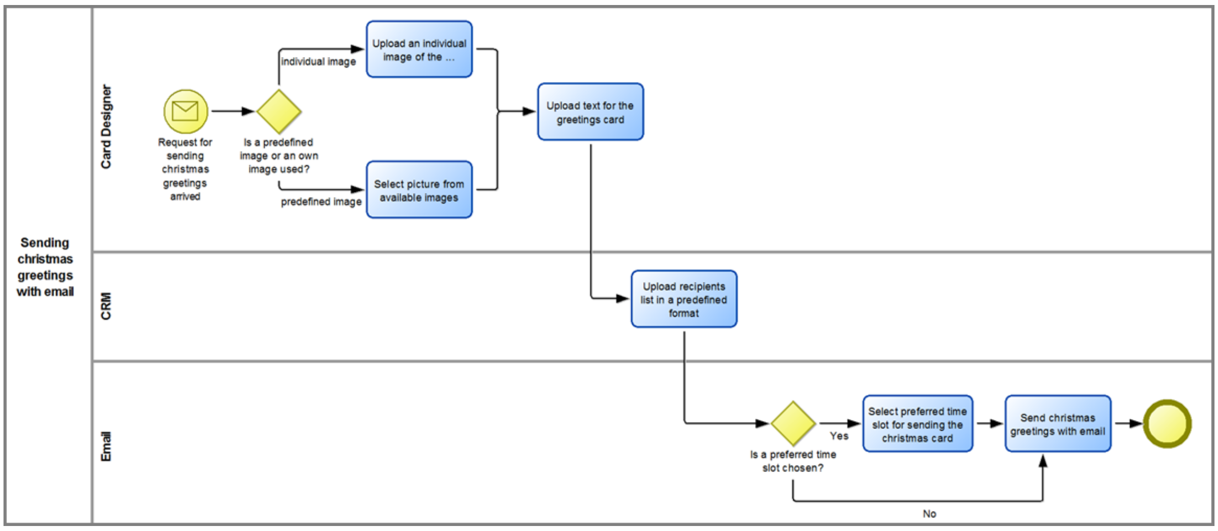

Sending christmas greetings with email
Card Designer
CRM
Email
Request for sending christmas greetings arrived
Is a predefined image or an own image used?
individual image
predefined image
Upload an individual image of the ...
Select picture from available images
Upload text for the greetings card
Upload recipients list in a predefined format
Is a preferred time slot chosen?
Yes
No
Select preferred time slot for sending the christmas card
Send christmas greetings with email

Figure 2 - Sending Christmas Greetings Business Process

As the ChristmasCardDesigner is a software component and hence orchestrated via the Cloud Provider Engine, a set of technical requirements have to be ensured to run this service. Technically, the ChristmasCardDesigner service is a Java based servlet, running inside a Java Application Server. Hence, this service fits as an example for a software component deployed on IaaS or PaaS. A simplified example set of technical requirements on IaaS and PaaS level is shown in Figure 3, including the underlying resources and the actual software stack to run the service.

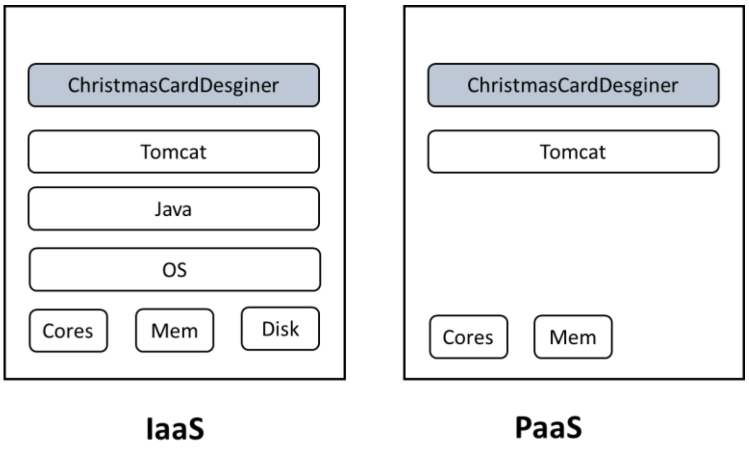


Figure 3 - IaaS/PaaS requirements for the ChristmasCardDesigner

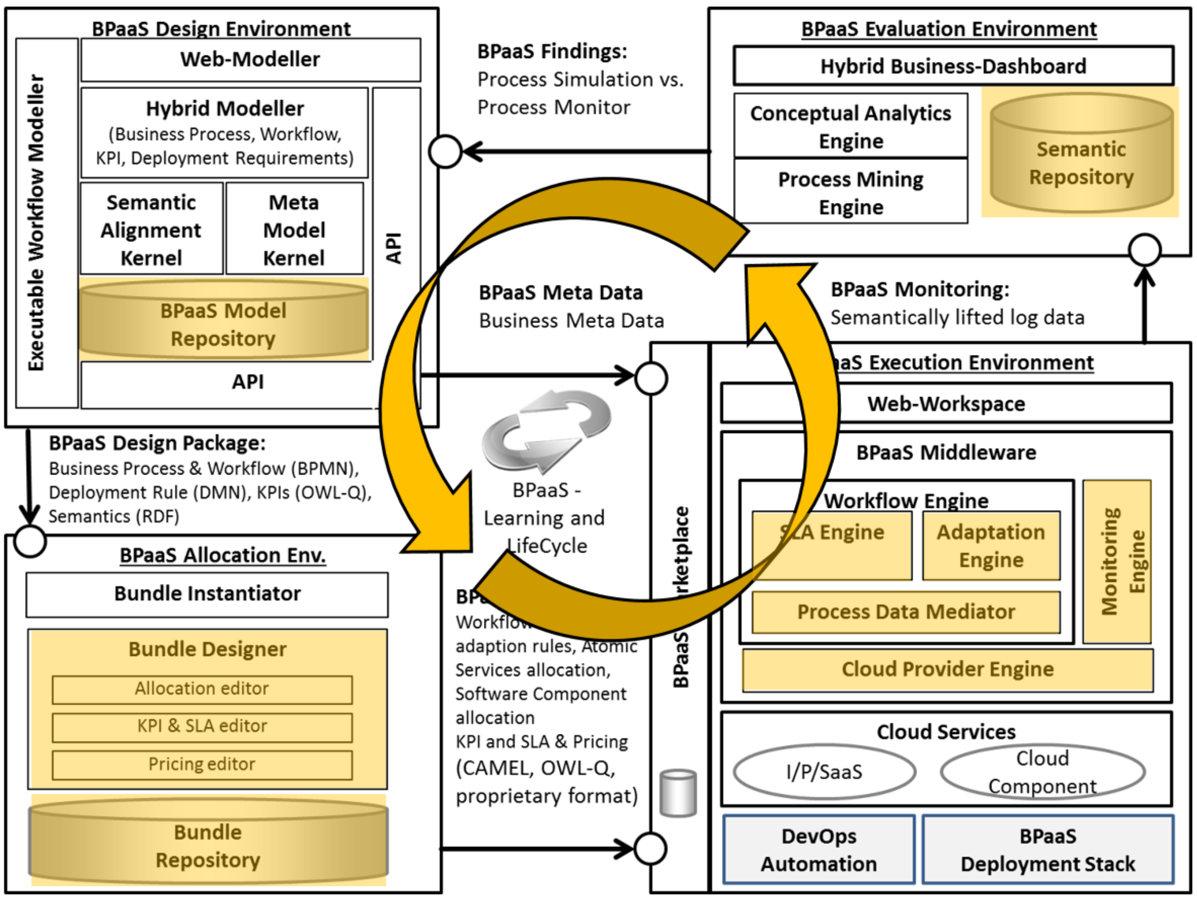
With further elaboration, we also show that such a component is also amenable for being adapted according to different scenarios based on the respective broker requirements posed.

1.3 Structure

The prototypes for BPaaS modelling are described in Section 2. This comprises the CAMEL and the OWL-Q extensions. Section 3 contain the prototypes concerning the Allocation Environment, i.e. the Smart Service Discovery and Composition, and the DMN-to-CAMEL-mapping. Prototypes for the Execution Environment that were developed for the cross-level management of cloud applications during run-time are described in Section 4. Section 5 concludes this deliverable with a summary and a supply of near future work directions.

2 BPAAS MODELLING PROTOTYPES

The prototypes concerning the BPaaS modelling are relevant across all Environments of CloudSocket (see Figure 4), since the specified models are used throughout most of the components. A modeller describes a certain service, which is later enriched with annotations that are also used in the adaptation phase, and subsequent in the execution phase. This section introduces modelling prototypes concerning the definition of cloud applications on different cloud layers and its behaviour in terms of adaptation to a certain context on run-time.



BPaaS Design Environment
Web-Modeller
Executable Workflow Modeller
Hybrid Modeller (Business Process, Workflow, KPI, Deployment Requirements)
Semantic Alignment Kernel
Meta Model Kernel
API
BPaaS Model Repository
API
BPaaS Findings: Process Simulation vs. Process Monitor
BPaaS Evaluation Environment
Hybrid Business-Dashboard
Conceptual Analytics Engine
Process Mining Engine
Semantic Repository
BPaaS Meta Data Business Meta Data
BPaaS Monitoring: Semantically lifted log data
aaS Execution Environment
Web-Workspace
BPaaS Middleware
Workflow Engine
SLA Engine
Adaptation Engine
Process Data Mediator
Monitoring Engine
Cloud Provider Engine
Cloud Services
I/P/SaaS
Cloud Component
DevOps Automation
BPaaS Deployment Stack
BPaaS Design Package: Business Process & Workflow (BPMN), Deployment Rule (DMN), KPIs (OWL-Q), Semantics (RDF)
BPaaS - Learning and LifeCycle
BPaaS Allocation Env.
Bundle Instantiator
Bundle Designer
Allocation editor
KPI & SLA editor
Pricing editor
Bundle Repository
BP Workflow adaption rules, Atomic Services allocation, Software Component allocation KPI and SLA & Pricing (CAMEL, OWL-Q, proprietary format)
BPaaS rketplace

Figure 4 – Affected components by the BPaaS modelling prototypes.

2.1 PaaS/SaaS support of CAMEL

The cloud computing stack comprises currently three main levels: IaaS, PaaS and SaaS. Two of them, namely IaaS and SaaS, have been already accommodated in the previous versions of CAMEL. However, PaaS gains a significant momentum lately due to the advantages that it enables users to focus on the development and provisioning of the core application functionality without requiring to deal with any information regarding the underlying infrastructure. To this end, to also enable exploiting this cloud service type, a PaaS extension in CAMEL was designed and implemented. This extension was a result from the study that was performed in the context of the previous deliverable, D3.3, as well as the review over the state-of-the-art and the corresponding meta-models and modelling languages that have been proposed.

The PaaS CAMEL extension was based on the following principles:

(a) same modelling style should be preserved,
(b) backward compatibility with respect to previous CAMEL versions should be guaranteed as much as possible.

These principles reduce the learning curve of the modeller while they map to the least possible changes in the implementation code of the system exploiting CAMEL with respect to the previous modelling features of this language. As such, the code developer can focus mainly on extending the code to exploit the new PaaS-related modelling feature of CAMEL.

The PaaS CAMEL extension focuses on the following aspects:

(a) description of requirements on PaaS services;
(b) description of PaaS types and instances mapping to certain PaaS capabilities;
(c) capabilities to configure the lifecycle of a component via a PaaS API.

Inline to this extension, the component description in CAMEL was slightly improved to enable incorporating accordingly the main PaaS notions as well as mapping to the respective components that can be covered by a PaaS. In the following, we describe the extensions made according to the above aspects and respective component enrichment. The class diagram which depicts this extension is shown in Figure 5.

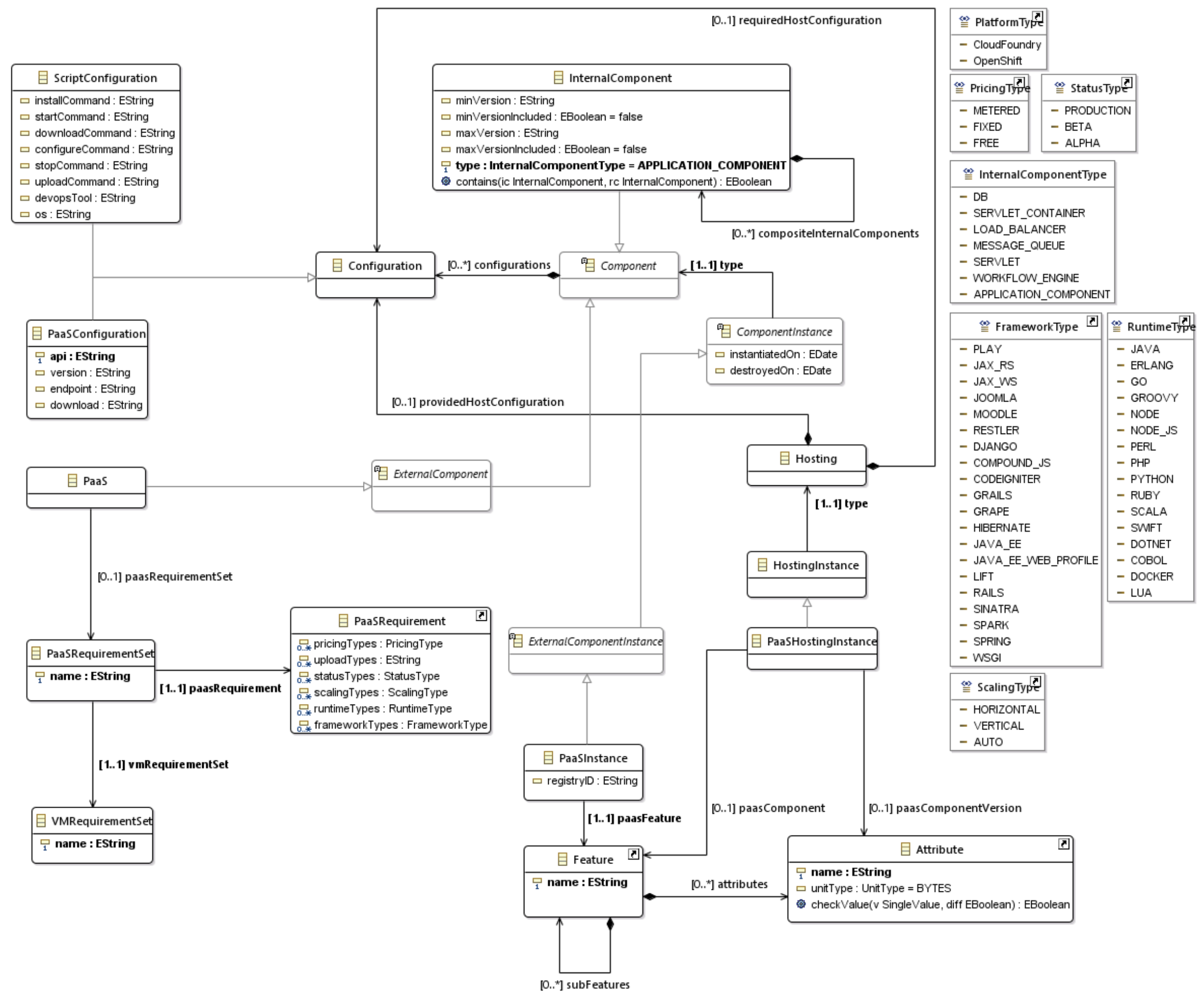


Figure 5 - The deployment meta-model class diagram focusing on the PaaS extension

To incorporate the inclusion of main PaaS notions and still follow the type-instance pattern, the *ExternalComponent* and *ExternalComponentInstance* classes (see also SaaS extension in D3.3) where sub-classed with respective PaaS-related classes, namely *PaaS* and *PaaSInstance*. This is due to the fact that a PaaS is considered as an external component with respect to the actual user application which is exploited to provide hosting and runtime capabilities. The actual software that is deployed on a SaaS component is again an internal component as it is part of the application. To reduce the modelling effort to the minimum, for each new PaaS-related class, the least possible information to characterise it has been generated. A *PaaS*, similarly to the *VM* class, is related to a set of platform and infrastructure requirements, named as *PaaSRequirementSet*, which need to be satisfied and acts as a placeholder for a PaaS component that provides a hosting port via which other components can be actually generated and hosted. This means that, e.g., DB and servlet container components could be hosted by that component, where such a hosting maps to respective templates of components that can be instantiated and run in the Cloud which relate to the actual components needed by the user application. For such components, configuration information is not needed, in the sense of running scripts, in order to install and run the components. A PaaS should be able to cater for this, provided that the respective needed component information is specified in CAMEL. More details about this will be supplied later on when the notion of a *PaaSInstance* is analysed.

We should also highlight here that we have also included a modification to the ProviderRequirement which is included in the hardware requirement set (*VMRequirementSet*) of a *PaaSRequirementSet*. This modification relates to the capability to either specify a concrete set of cloud providers to be used for the IaaS/PaaS selection & respective hosting or an indication about the type of the cloud (provider) (i.e., private or public). In both cases, we actually reflect a selection over either PaaS and/or IaaS services depending of course on the deployment (and other types of) requirements of the corresponding application. The type of a cloud provider is specified by explicating the respective member of a newly introduced enumeration called *ProviderType* in the requirement package/sub-DSL.

The *PaaSRequirementSet* should be used in an equivalent way as in the case of *VMRequirementsSet*. In particular, a global *PaaSRequirementSet* can be posed at the deployment model level that will hold for all the *PaaSe*s that are defined in this model. Moreover, a (local) *PaaSRequirementSet* can be linked to a *PaaS* to specify local PaaS requirements. Both requirement types (local and global) can be exploited in conjunction or even independently. A *PaaSRequirementSet* is associated to an actual *PaaSRequirement* and an *IaaSRequirementSet* to reflect that it represents both platform and infrastructure-related requirements. The former places requirements on the actual PaaS characteristics, while the latter, on the corresponding IaaS characteristics which can be exploited and offered under this PaaS.

Various PaaS characteristics have been modelled, basically inspired by the selection criteria in paasfinder.org. These criteria include:

(a) the *platformTypes*, where we can basically see that most of the PaaS exploit either `OpenShift` or `CloudFoundry`,
(b) the *runtimeTypes*, where a vast variety of runtimes has been considered, such as `Java` and `Cobol`,
(c) the *frameworkTypes*, such as `Java EE` or `Play`,
(d) the *scalingTypes* where different types of scaling can be supported from `manual` (either horizontal or vertical) to `automatic`,
(e) the *pricingTypes* which could be `metered`, `fixed` or `free`,
(f) the *statusTypes* explicating the status of the respective API offered in terms of either being in `production` or in `alpha` or `beta` versions and finally
(g) the *uploadTypes* which cover the way uploading can be supported, including `git`, `maven` or `gradle`.

The multiplicity of all these properties is from 0 to * indicating that the modeller can skip one criterion or provide multiple values for it. However, the semantics can be different depending on the respective attribute at hand. Multiple values for attributes like *pricingTypes* means that the modeller imposes a disjunctive constraint on these values. On the other hand, for attributes like *runtimeTypes*, the semantics is that all respective values should be covered (conjunctive constraint).

Before entering details about the instance level, we need to highlight that the *InternalComponent* was modified to include a new enumerated attribute called *type* which provides insight about the actual kind of the component. The following members of the *InternalComponentType* enumeration are envisioned for now: DB, MESSAGE_QUEUE, LOAD_BALANCER, WORKFLOW_ENGINE, SERVLET_CONTAINER, SERVLET and APPLICATION_COMPONENT. The first five members map to respective services offered by a PaaS. In this sense, internal application components with such types could be hosted by a PaaS. The SERVLET member maps to the *InternalServiceComponent* class as it indicates an internal software component offered as a service. In this sense, instances of the latter class will have their type fixed to SERVLET. In case that an internal component does not map to the first six types, then it is considered as a normal software component which should be just installed and run, without actually being offered as a service. In this case, the type of such a component should be APPLICATION_COMPONENT, which also represents the default enumeration value for the *type* attribute.

InternalComponent was also extended to incorporate versioning information. In particular, we now allow to indicate what is the minimum and maximum version of a component. Such information is not needed in case of normal application components (i.e., mapping to the APPLICATION_COMPONENT type). However, in case of components covered by a PaaS, we need to have the name and the version range for that component. Otherwise, we would not be able to find and select the respective service capability provided by the PaaS to be selected. As such, this name and versioning information is considered as a kind of software requirement over the respective PaaS capability, thus influencing the selection of a PaaS. While name information is already covered by the *Component* class, versioning information for components was captured via the introduction of the *InternalComponentRequirement* class in the requirement package of CAMEL. An internal component is also associated to 0 or 1 instances of this class to enable mapping it to its respective versioning requirement. This class, apart from capturing the actual range limits of the component version, includes two Boolean attributes, which explicate whether these limits should be included or not. This enables us to perform a more precise search over the PaaS capabilities. Let us now provide a concrete example for this extension. For instance, in case of a Tomcat servlet container, it is important to know the name of the component (i.e., Tomcat) and the respective range needed by the user, such as [6.0,7.0). This can allow us to match and select, e.g., the Anynines PaaS[2] which enables exploiting a tomcat container with a version between 6.0.* and 7.0.*.

Entering now the instance level, a *PaaSInstance* represents a certain instance of a PaaS that maps to a particular PaaS provider. The latter mapping is established by referring to the respective feature of the CAMEL provider model of this provider. Moreover, a *PaaSInstance* is also characterised by a *registryID* to enable the respective environment or component exploiting the CAMEL model to retrieve information about the respective PaaS from the corresponding entry in the Registry.

The special types of components that need to be hosted by a PaaS should be properly configured. However, this must not be performed at the type level where the respective software requirement has been provided. It can only be applied at the instance level, when the most suitable PaaS has been selected that fulfils all the deployment requirements set. However, when specifying the *PaaSInstance*, it cannot be directly related to the component instances that it should offer. This is only performed via the *HostingInstance* class. Thus, based on this analysis, the latter class seems to be the most suitable place to enforce the mapping between the needed component

[2] European Cloud Foundry Platform, https://paas.anynines.com/

(instance) and the respective PaaS capability. To preserve CAMEL modelling style as well as guarantee backwards compatibility, it was decided to sub-class the *HostingInstance* class with the *PaaSHostingInstance* one. Now the latter class includes an association to the respective PaaS feature and an attribute-value pair that reflects the concrete desired capability. For instance, by continuing the example of the servlet container component and supposing that this component has one instance to be hosted by a respective PaaS instance, a *PaaSHostingInstance* will be created that will refer to the component instance's required hosting port and the PaaS instance's provided hosting port. This *PaaSHostingInstance* would map to the "Tomcat" feature (which would be a sub-feature in the feature hierarchy of the PaaS feature) in the respective provider model of the provider offering the PaaS (instance) as well as to its *version* attribute imposing the value of "6.5". In this sense, the hosting instance will denote in the end that the PaaS instance should host a tomcat component with 6.5 as its version.

The last modification in CAMEL for the PaaS extension concerns updating the configuration of the lifecycle of internal components to exploit additional possibilities which come via the use of DevOps tools as well as PaaS APIs. The extension is aligned with the PaaS extension proposal in D3.3 but slightly modifies it by relying on the following assumptions:

(a) DevOps tools allow to install modules onto the local system but still requires to have configuration commands in place;
(b) the PaaS APIs are heterogeneous and usually require an own model-based description of the application/component and the environment.

As such, we came up with the decision to:

(a) not explicitly model DevOps configurations as classes as they are just special instances of script-based configuration. To this end, we have created one class named as *ScriptConfiguration* which includes references to all possible lifecycle commands as well as to the respective OS for which this configuration can be applied. To also cater for DevOps-based configurations, an additional field was incorporated in the same class to denote the name of the DevOps tool to be exploited;
(b) the *PaaSConfiguration* is modelled separately, thus requiring also to create a common super class for PaaS and script-based configuration called *Configuration*. This means that the previous version of the *Configuration* class becomes *ScriptConfiguration* and a new (abstract) class is generated with the same name that does not incorporate any special information. The actual content of *PaaSConfiguration* is platform independent. As such, we abstract away from information that could include platform-specific configuration directives which depends on the actual API being offered by the PaaS provider.

To be as generic as possible, the only information that has been covered for a PaaS configuration is the following:

(i) the actual API as a String to be exploited,
(ii) the version of this API,
(iii) the endpoint of the API, if it is external to the actual running platform, and
(iv) the download command for the application and environment models which should conform to the API-specific meta-model.

Apart from the first attribute, which is obligatory, the rest are considered optional. This is especially true for the last one as we can also consider that the respective provisioning platform that should exploit the CAMEL model should be able to derive a respective API-specific model for the components and environments concerned out of it. As such, we avoid enforcing modellers to perform similar modelling tasks twice but allow them to concentrate on and finalise the deployment model of their application.

2.1.1 Example

By relying on the well-known Christmas Card BPaaS use case, we now provide a specific example of how the respective CAMEL model could be specified. Suppose that the Card Designer component needs to be deployed over a (public) PaaS which needs to support the CloudFoundry PaaS platform and metered-based charging. The Card Designer is a java-based application component that needs to be hosted as a service by a respective servlet container, like Tomcat. In addition, due to major location constraints of the customers of the broker, the BPaaS should be deployed in Europe. By considering the current capabilities of the existing PaaS providers, the respective PaaS which best satisfies the user requirements could be found via employing a PaaS discovery service. Such a service could take the form of a PaaS service discovery algorithm (a potential extension of the algorithm proposed in Section 3.1) or an external PaaS search engine, like paasify.it. In any case, the set of required features for the respective BPaaS could lead to the situation where a few or even one PaaS service could be exploited, namely the Atos Cloud Foundry. In this case, the respective CAMEL model would take a particular form, which is shortly analysed in the following while a detailed description of the model in XMI form is provided in the appendix of this deliverable.

By considering the original IaaS-based deployment model for the Card Designer component, some specific differences can be observed:

(a) Now the deployment model comprises two components, instead of one, as the tomcat container (with type SERVLET_CONTAINER) is also explicitly designated to be offered by the respective PaaS service.
(b) The Tomcat container is hosted by the selected PaaS while the actual Card Designer component is hosted in turn by this container.
(c) A PaaS-based configuration is given for the corresponding Card Designer component.
(d) The PaaS node/type in the CAMEL deployment model is associated to a respective set of requirements. These requirements include:
    i. a PaaS requirement which involves PRODUCTION as *statusType* for the PaaS, METERED as its *pricingType*, CloudFoundry as its platformType, and Java as its runtimeType;
    ii. a location requirement which indicates that the location of the PaaS should be in Europe and
    iii. a provider requirement indicating that the provider type should be PUBLIC.

The respective visualisation of this example focusing on the type level is shown in Figure 6 while the whole CAMEL model of this example is provided in the appendix.

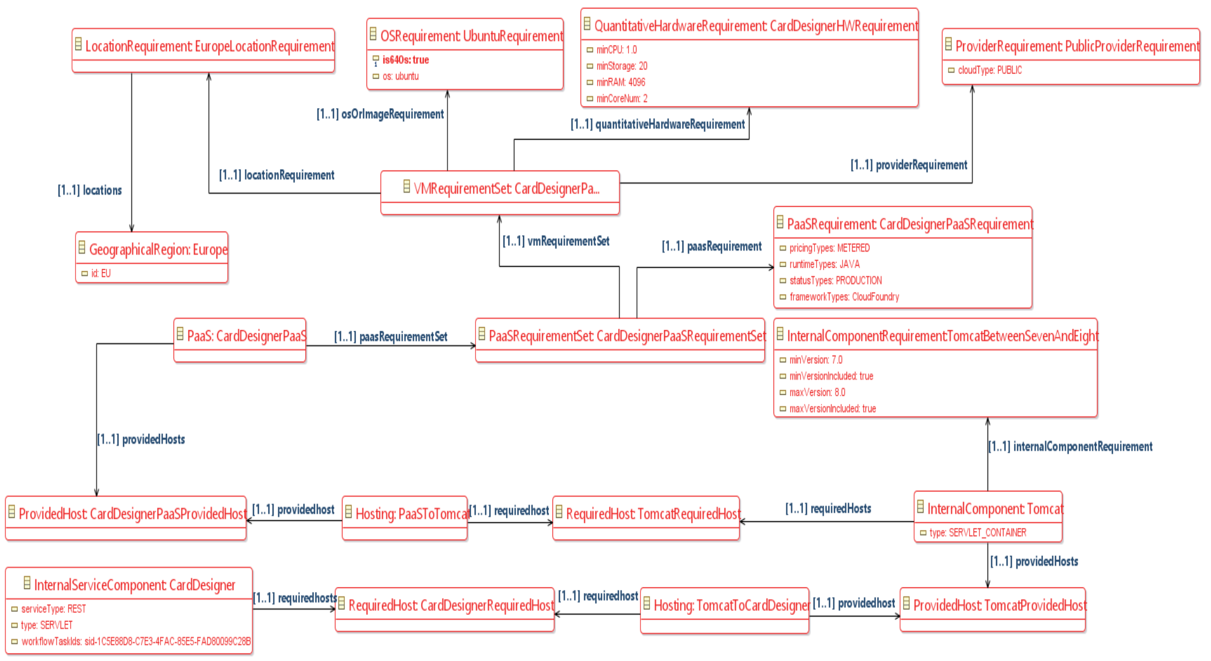

LocationRequirement EuropeLocationRequirement
OSRequirement UbuntuRequirement
is640s: true
os: ubuntu
QuantitativeHardwareRequirement CardDesignerHWRequirement
minCPU: 1.0
minStorage: 20
minRAM: 4096
minCoreNum: 2
ProviderRequirement PublicProviderRequirement
cloudType: PUBLIC
[1..1] osOrImageRequirement
[1..1] quantitativeHardwareRequirement
[1..1] providerRequirement
[1..1] locations
[1..1] locationRequirement
VMRequirementSet CardDesignerPa...
GeographicalRegion: Europe
id: EU
[1..1] vmRequirementSet
[1..1] paasRequirement
PaaSRequirement CardDesignerPaaSRequirement
pricingTypes: METERED
runtimeTypes: JAVA
statusTypes: PRODUCTION
frameworkTypes: CloudFoundry
PaaS: CardDesignerPaaS
[1..1] paasRequirementSet
PaaSRequirementSet CardDesignerPaaSRequirementSet
InternalComponentRequirementTomcatBetweenSevenAndEight
minVersion: 7.0
minVersionIncluded: true
maxVersion: 8.0
maxVersionIncluded: true
[1..1] providedHosts
[1..1] internalComponentRequirement
ProvidedHost CardDesignerPaaSProvidedHost
[1..1] providedhost
Hosting: PaaSToTomcat
[1..1] requiredhost
RequiredHost TomcatRequiredHost
[1..1] requiredHosts
InternalComponent Tomcat
type: SERVLET_CONTAINER
[1..1] providedHosts
InternalServiceComponent CardDesigner
serviceType: REST
type: SERVLET
workflowTaskIds: sid-1C5E88D8-C7E3-4FAC-85E5-FAD80099C28B
[1..1] requiredhosts
RequiredHost CardDesignerRequiredHost
[1..1] requiredhost
Hosting: TomcatToCardDesigner
[1..1] providedhost
ProvidedHost TomcatProvidedHost

Figure 6 - CAMEL fragment focusing on the type level of the PaaS deployment example

2.2 SLA support of OWL-Q

The SLA OWL-Q extension, named as Q-SLA, has been extensively analysed in D3.3 [1] while it has resulted in a particular publication [4]. To this end, the goal of this section is to summarise the main features of the language as well as provide useful implementation details enabling the exploitation of this language even outside the context of this project.

2.2.1 Features

The main features of Q-SLA can be summarised in the following list:

- Relies on OWL-Q and thus inherits its excellent coverage of all measurability aspects (e.g., metric, unit and value type).
- QWL-Q and this Q-SAL are semantic languages enabling the syntactic, semantic (based on rules) and constraint-based validation of SLA models.
- Enables the participation of third (non-signatory) parties in the SLA and the assignment of respective duties on them (e.g., monitoring of SLO metrics, evaluation of SLOs).
- Enables the specification of both Service Levels (SLs) and Service Level Objectives (SLOs), where SLs can be considered as a logical combination of SLOs.
- Enables the dynamic transitioning of SLs to cater for the following situations: (a) cover the low-level SL delivered during maintenance periods; (b) allow a signatory party to transition from one SL to another one when a change of requirements (e.g., service customer needs to address now a greater number of clients) or another kind of condition occurs (e.g., percentage of violations within a certain period is above a threshold).
- Enables the specification of both rewards and penalties to hold when the promised SLO is either surpassed or violated, respectively.

- Covers the modelling of the service price model which is connected also to the rewards and penalties specified.
- Enables the capturing of critical situations which require a special handling over the SLA (e.g., renegotiation or cancelling). Both the conditions mapping to these situations as well as the handling actions can be specified in this case.
- Supports the specification of hierarchical SLAs through the modelling of parent-child relationships between SLA contracts.

2.2.2 Implementation

This OWL-Q extension has been fully realised as a sub-facet of the original specification facet. To this end, a corresponding OWL file has been created mapping to this sub-facet in which all the features have been realised by building on top of the elements involved in the specification and the other OWL-Q facets. SLAs can now be specified by using any ontology editor, such as Protege[3]. To facilitate this specification, a mid-level ontology has been constructed based on OWL-Q which includes a basic set of domain-independent non-functional terms, such as quality attributes and metrics. Therefore, such terms can actually be re-used in the specification of the respective SLOs (which map to conditions over such non-functional terms).

A Q-SLA/OWL-Q (bidirectional OWL-to-Java code) parser is underway in order to enable the programmatic specification of SLAs. This can enable not only building any kind of editor on top of OWL-Q but also on the fly generation of SLA content via automatic programs. Such a feature could be quite useful in the context of negotiation where automated agents need to negotiate the SLA terms and thus be able to modify SLAs on demand according to the negotiation strategies of the participants for whom they act on behalf.

2.2.3 Set-Up

The whole OWL-Q specification, including the SLA extension, and the respective OWL-Q parser are available in the UULM's git repository[4]. The whole documentation of the latter component will also be made available in the project wiki[5].

The parser component will map to a Java maven project. In this sense, one can just clone the respective git repository and involve usual maven commands to compile as well as run the parser. Of course, we foresee mainly the use of the parser inline in another component. As such, mainly the compilation of that component (via "mvn clean install" command) will be mostly relevant.

2.2.4 Future Work

2.2.4.1 SL Transitioning

Currently, Q-SLA enables the transitioning over whole SLs. In the near future, we will examine whether it makes sense to enable a more fine-grained transitioning at the SLO level. This could be explored by also considering the CloudSocket use cases to enable a respective validation of the corresponding modification necessity. Once this is done and the validation is positive, then we will of course implement the required modification.

2.2.4.2 SLA Composition

Q-SLA is able only to specify SLA hierarchies by connecting them via parent-child relationships. This can be considered as basic support to SLA composition. In the near future, we plan to extend Q-SLA to be able to specify pure SLA compositions in which respective dependencies between non-functional terms in different SLA

[3] http://protege.stanford.edu/
[4] https://omi-gitlab.e-technik.uni-ulm.de/cloudsocket/owlq_parser
[5] https://www.cloudsocket.eu/common-understanding-wiki/-/wiki/Main/OWL-Q+Parser

levels can be expressed. In this way, Q-SLA will certainly become a SLA language which provides the best possible support to all activities in the SLA/(cloud) service lifecycle.

2.3 SRL update on CAMEL

2.3.1 SRL Update Analysis

The Scalability Rule Language (SRL) in CAMEL has focused only on scaling issues based on the respective requirements that it had to cover in the PaaSage project. However, by considering the feedback that has been obtained in that project as well as the context of the CloudSocket project and the requirements that it brings about (e.g., support cross-layer adaptation, cover additional adaptation actions at different layers), it was decided to evolve the scaling package of CAMEL in order to transform it into a full-fledged adaptation DSL which covers the specification of cross-level and advanced adaptation rules thus further advancing the state-of-the-art in cloud adaptation modelling.

The SRL update analysis starts with explicating the main drivers for the scaling package evolution via the supply of three main adaptation scenarios and then we proceed with the actual detailed description of this package evolution. Please consider that CAMEL and especially the scaling package has been shortly analysed in D3.3 [1] so there is no point in repeating the same information in this deliverable.

The following three main adaptation scenarios are now covered by this CAMEL extensions:

- *cloud bursting*: this is a requirement from PaaSage which is not captured in SRL as in PaaSage, a different intermediate coverage of this scenario is attained via dynamic application reconfiguration (e.g., new hosting of a component in a public cloud VM where the VM and of course the components are already described in the deployment model). However, in the case of the CloudSocket project, there is no explicit deployment reconfiguration phase which involves a certain reasoner component that implicitly can detect the need of introducing a new component hosting. On the contrary, a rule-based approach is followed which requires explicitly specifying the respective adaptation action that has to be performed. As such, in the case of CloudSocket, we need to define an adaptation action which introduces the hosting relationship between the application component and respective public cloud VM.
- *multi-component scaling*: this is again a requirement originating from PaaSage which depends on the level of deployment granularity (single component per VM or multiple components per VM). In particular, it concerns the fact that within a particular scaling it is not certain which components from an existing VM should be scaled. Moreover, this is not apparent from the semantics of the component description in CAMEL. We could take the following directions to realise this: (a) communication requirements could signify the needed semantics such that when two components need to communicate locally, then both have to be scaled and not just one; (b) we consider each component independent from the other and explicitly state which components from a respective VM have to be scaled. Via (b) we could consider a case where the local communication semantics might need to be broken to better cover the respective scaling requirements.
- *cross-layer adaptation*: based on the need in CloudSocket to support cross-layer adaptation which involves the orchestrated execution of adaptation actions at different layers, CAMEL should be extended with the capability to specify a whole adaptation workflow comprising multiple adaptation actions in different layers. By assuming that the event pattern detection is already well covered in the context of scaling / adaptation rules, then this extension along with the requirement to cover well the description of individual adaptation actions at different layers have to be accommodated in this scaling package evolution.

By considering the above three main adaptation scenarios to be supported, the respective modifications performed in the CAMEL scaling package, which are depicted in Figure 7, were the following:

- Mapping a horizontal scaling action to multiple components to enable scaling multiple components per VM. In this way, we can actually cover three possible cases: (a) single component scaling per VM; (b) multiple component scaling per VM; (c) combinations where each scaling action even in the context of one VM is mapped to a different adaptation actions. In this way, in this latter case, we can scale two components together in one VM as well as scale separately the third remaining component in a new instance of this VM. The respective limits of scaling map to the component level so there is no need to further update this - in case that we need to scale one or more components where the scale limit for one of these components has been reached, then the scaling action cannot be completed.
- Renaming of some classes was performed to denote the change of scope of this package as well as a certain naming pattern (mapping to the fact that if something is a task or an action, the "task" postfix in its name does not have to be repeated). To this end, the *ScalabilityModel* became *AdaptationModel*, the *ScalingAction* became *Scaling*, the *HorizontalScalingAction* became *HorizontalScaling*, and the *VerticalScalingAction* became *VerticalScaling*.
- Removal of *ActionType* enumeration as it is redundant. Decided to create specific classes which map to all possible individual adaptation actions, including new ones like service replacement and migration of VMs/components. As this enumeration is also needed in the specification of organisational permissions, only a small part of it was moved to the organisation package. This part has been named as *PermissionActionType* and it now includes only two members, the *READ* and *WRITE* action types.
- Right part of scalability rule should now map to the specification of a workflow of adaptation actions. Please find in the sequel the way this has been modelled.

Apart from renaming ScalabilityRule to AdaptationRule to designate the extension of its applicability, such a rule is now associated to one *AdaptationTask*, i.e., any kind of adaptation task / action which is able to represent either atomic adaptation tasks or whole adaptation workflows. To this end, this latter class has been split into two main subclasses: *SimpleAdaptationTask*, which signifies an atomic adaptation task, and *CompositeAdaptationTask*, which signifies a composite adaptation task or workflow. The *AdaptationTask* is actually a renaming of the *Action* class, while it has been made abstract and is now linked to the respective responsible which should perform the corresponding adaptation task (which should usually be a service or any kind of web-based component which can be informed for the need to perform the action and is of course able to execute it). In order to cover the case of an adaptation task failure, an *AdaptationTask* is associated to a recovery workflow. This mapping is covered by associating an adaptation task to another adaptation task that represents the recovery workflow to be executed.

The *CompositeAdaptationTask* represents an adaptation workflow. This class is associated to the set of sub-tasks to be executed by this adaptation workflow and can be categorised in turn into more concrete classes, namely *SequentialAdaptationTask*, *ParallelAdaptationTask*, *ConditionalAdaptationTask* and *SwitchAdaptationTask*. The *SequentialAdaptationTask* and *ParallelAdaptationTask* represent workflows which execute their sub-tasks in sequence or in parallel, respectively. As such, their name and the respective association to their sub-tasks (with the corresponding reference order) are enough to denote their semantics. The *ConditionalAdaptationTask* is a conditional workflow with the semantics of applying a specific event out of which one from two alternative adaptation tasks can be selected: the first task is selected when the event does occur while the second task when this event has not actually occurred. As such, this kind of composite adaptation task maps to one Event, whose specification is already covered in the adaptation package/meta-model, and needs to be associated to only two sub-tasks whose order does play a role (remember first task maps to event occurrence). Finally, the *SwitchAdaptationTask* maps to a workflow which applies a switch kind of statement over the respective adaptation possibilities. In this respect, we need to evaluate such statement over a dynamic variable which can take multiple values instead of just two as in the case of a condition (in

ConditionalAdaptationTask). In this sense, the most logical selection of the respective construct for representing such a dynamic variable is the *MetricFormulaParameter* which can represent both metrics and formulas over such metrics. Apart from this parameter kind, a *SwitchAdaptationTask* is associated to a list of *ValueToTask* elements which denote the mapping from a value of the metric formula parameter to a respective adaptation task (only from the list of sub-tasks of the composite adaptation task).

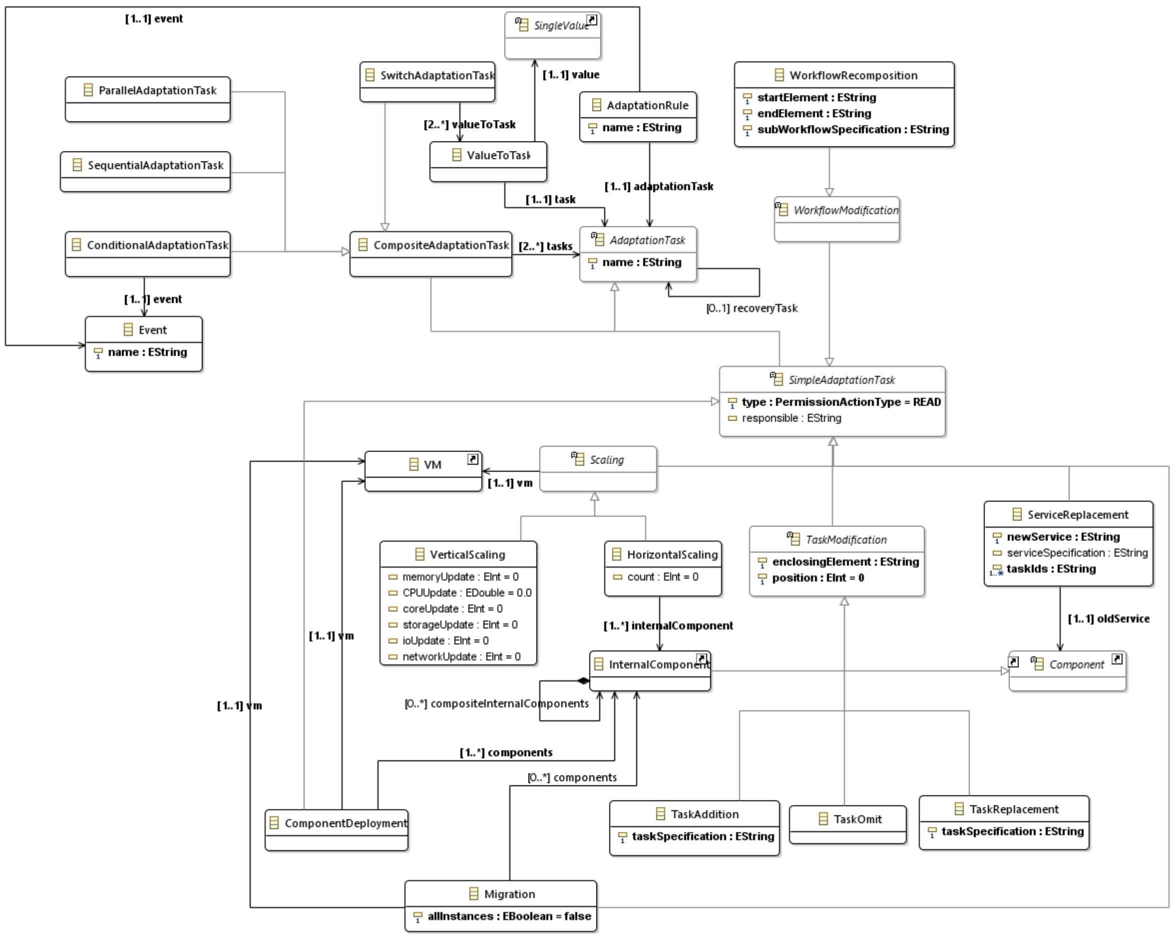


Figure 7 - Snapshot of CAMEL focusing on new and updated classes in the adaptation/scaling package

As already indicated, a SimpleAdaptationTask represents an individual adaptation task. As this class is abstract, it has been accompanied with the specification of respective sub-classes which map to concrete individual adaptation actions that can be performed on the IaaS, SaaS & WfaaS levels, thus also covering two new levels. In particular, we have developed the following new classes: ComponentDeployment, Migration, ServiceReplacement, TaskModification, WorkflowModification, TaskAddition, TaskOmit, TaskReplacement, WorkflowRecomposition, EventCreation and Reporting.

ComponentDeployment concerns the deployment of a component in a VM. This is different from the scaling as the component and/or the VM might not be part of the original and applicable deployment model of the application / workflow. The latter means that there can be components and VMs that might be described but are not connected to each other, i.e., a component is not hosted on any other component and the VM does not host any component. In this case, the goal of this class is to tie these two component types together and enable the respective creation of the component instances and the enforcement of their relationship. The class, thus, includes mainly two associations: one to the (internal) component to be deployed and one to the VM on which the component will be hosted. Here the realisation of the component deployment adaptation action is implied to include two main steps: (a) reasoning over which VM flavour to select for the respective component (based on the respective requirements posed), in case this VM has not been reasoned before (the VM is not used in other component hostings provided that the component is not associated to specific requirements that can have an effect over the VM flavour selection); (b) actual component deployment via the respective cloud orchestration framework. Please note here that we consider that both the component and the VM are described in the deployment model and are thus only associated to this adaptation task.

The incorporation of this class could provide support to cloud bursting scenarios (initial rule to start the deployment of an existing component to a certain (public) VM, subsequent rule could also handle the scaling of this components in the public cloud, if needed) as well as the incorporation of new component deployments in the currently running deployment model of an application. For instance, there can be a case where a scaling of a component needs to be accompanied with the addition of a *LoadBalancer*. The latter component could be part of the original deployment model at the type level but not part of the actual concrete deployment of the application. The introduction of this component could cater cases where the load between the old and previous instances of the component scaled needs to be balanced.

Migration concerns the migration of one or more components from an old VM to a new one. To specify this, three associations have been created: (a) one to the old VM; (b) another to the new VM; (c) to the components to be migrated. The latter association is optional. In case it is absent, then this means that all the components hosted in a particular VM need to be migrated. Moreover, there is a boolean attribute which denotes the multiplicative semantics of the migration. In particular, when this attribute takes a true value, then this means that all instances of the components hosted in all instances of the old VM should be migrated to new instances of the new VM; otherwise, only the current instance(s) affected are to be migrated to a new instance of the new VM. This distinction should of course be correlated to respective metrics which operate on either the type (i.e., class-based metrics) or the instance level (i.e., instance-based metrics), respectively.

ServiceReplacement concerns the replacement of one service with another one in the context of one or more (BPaaS) workflow tasks. To this end, the following pieces of information are specified in this class: (a) a reference to the service to be replaced. In this case, we refer to a *Component* in order to have the two possible cases mapping to the replacement of an internal service component or an external SaaS one; (b) a reference to the replacement service - here we have included two attributes that can be exploited only when we need to explicate exactly which service to use for the replacement. Otherwise, the system can internally identify which concrete service will be used for that replacement. These attributes are the *newService* and *serviceSpecification*. The first attribute can refer to the endpoint of the service or the id of the respective Registry entry or any other pointer that can identify properly the new service. To this end, it has been mapped to a String-based type. The second attribute has been included to cover the case of a service composition. In that case, additional information needs to be provided, mapping to the specification of the service workflow that has to replace the previous service. In this latter case, as the composition has been produced dynamically, there is no registry entry for the description of the composite service; (c) the ids of the tasks where the respective replacement will apply.

A *WorkflowModication* is an abstract class that denotes any kind of modification that can be performed on the overall workflow level. Currently, only one concrete subclass of that class has been generated which is named as *WorkflowRecomposition* with the main target to recompose a part of a specific workflow. In this sense, the workflow part is designated by two associations: one to the starting element and one at the ending element on which the replacement will take place. Another association is then used to retrieve the actual sub-workflow specification that will replace the existing one.

TaskModification is an abstract class and a sub-class of *WorkflowModification* which denotes any kind of modification that can be performed at the task level. This class is associated to the id of the workflow element that directly encloses the task to be modified as well as to the position in that element of the task at hand. This abstract class is sub-classed by three concrete classes, namely *TaskAddition*, *TaskReplacement* and *TaskOmit*. The first subclass represents the case of inserting a new task in the workflow at the designated position; thus this class is associated to the description of the task to be incorporated. The second subclass has the same content with the previous one but different semantics. In this case, the goal is to replace one task, which lies in a well designated position in the workflow, with another one whose specification is provided to be incorporated in the current workflow specification. The third sub-class represents the case that an existing task is omitted from the current execution of the running workflow (instance). In this case, no additional information needs to be provided, as the required information is already covered by the parent abstract class.

EventCreation represents a concrete action with the main purpose to create an event and report it in the measurement & evaluation system. This action is associated to the respective event to be created (which is specified naturally via SRL). Specific additional information communications might be implementation-specific and do not need to be provided at the current moment.

Reporting represents an action that reports a message to a component (e.g., a UI one). This could be considered as similar to *EventCreation* with the main exception that the content of the message can be arbitrary. In addition, here the recipient should also be clarified as well as the way to contact him/her in order to send the message. As such, the following information needs to be provided for this class: (a) *message*: the content of the message to be distributed as a String; (b) *recipients*: a list of Strings which could be interpreted as names of components or email addresses or any other kind of information that can distinguish the recipient position/address; (c) *protocol*: the actual protocol/means for communication which could be for example "email".

The overall set of modifications proposed on SRL make it an advanced adaptation language which is able to cover a plethora of adaptation scenarios. This language also inherits the main event combination mechanisms of the original SRL language that could represent the triggering conditions of these scenarios while also it is equipped with additional adaptation modelling capabilities that make it suitable to compose adaptation actions on different levels of abstraction. Of course, the SRL language is substantially extended and modified such that it is not totally compatible with its previous version. This is a small penalty we have to pay in order to make this language a truly adaptation one.

2.3.2 Examples

2.3.2.1 Cloud Bursting

Description. Suppose that we have still the Christmas Card Sending process where now both service components (card designing and email sending) are internal. In this case, bursting the card designer would not make sense as this component is not heavily used and does not seem to create a major utilisation issue on the respective VM on which it is deployed. On the other hand, if the same email service is used by the same client many times, there is of course the danger that this service can overload the respective VM. As such, it might be required to scale the latter service component by allocating yet another VM. However, such a scaling might not succeed if the private infrastructure on which both components have been deployed is already full. To this end, the email service component would need to be burst.

Concerning the original deployment model in CAMEL, the two BPaaS service components will be placed on different private cloud VMs which are named as CardDesignerVM and EmailServiceVM, respectively. To also support cloud bursting, a new VM named as EmailServicePublicVM is introduced which will map to certain requirements over a public cloud (i.e., those quantitative hardware requirements also prescribed in the case of EmailServiceVM) and will host a new instance of the email service component. One communication is specified in the original CAMEL deployment model: one from the card designer service component to the email service one.

To support cloud bursting, we rely on the following scenario: when the EmailServiceVM's average CPU is above 70% and the private cloud quota limit is reached, then a new instance of the email service component will have to be created and hosted in an instance of the EmailServicePublicVM while also a communication link between this new instance and the card designer instance will have to be created. Such an adaptation could be specified in the form of a component deployment adaptation task where the component is the email service one and the VM is the EmailServicePublicVM.

The architecture at the type level of the respective example is depicted in the following figure where we can clearly see the components utilised, the VMs that they can host them and the respective new hosting link that is to be created upon the email service bursting.

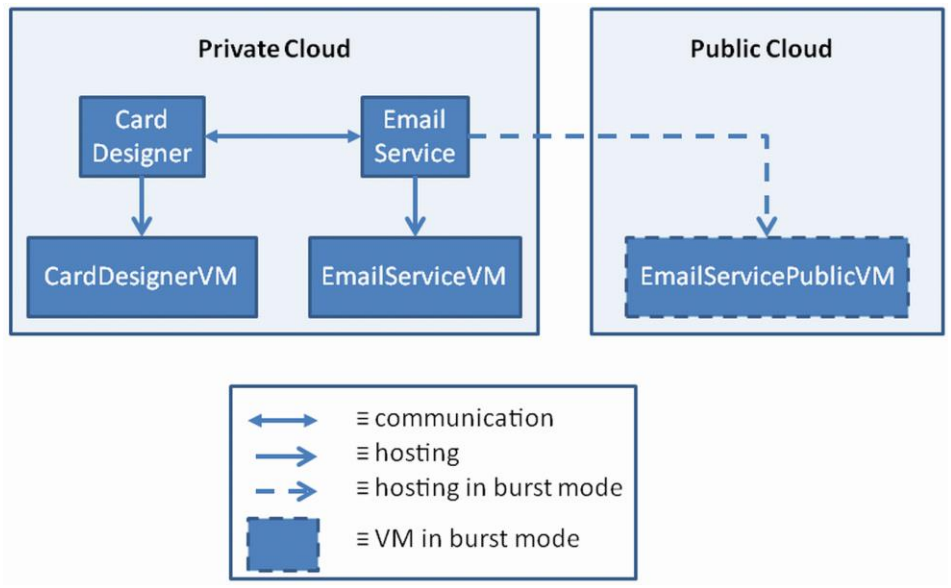


Figure 8 - The cloud bursting scenario for the Christmas Card Sending BPaaS

Implementation. A respective CAMEL fragment for this scenario is visualised in Figure 9, while the whole CAMEL model in XMI form is provided in the appendix of this deliverable.

2.3.2.2 Service Replacement

By considering yet the Christmas Card Sending BPaaS use case, we consider now a scenario where one service component is under-performing or becomes unavailable and we need to replace it with another one. We rely on the assumption that the email service has a response time of more than half of a minute which indicates that it is possibly overloaded. In this case, we consider that both the card designer component and the email service one have been deployed in one cloud, namely the Omistack cloud operated by UULM. We also assume that we have here two opportunities for service replacement: (a) the email service is replaced with another one; (b) another instance is exploited from the same service which has been created in the context of another client. In case of (a), the service interface can be altered. As such, we would need to also modify the BPaaS workflow, which is currently not supported by the CloudSocket research prototype. In case of (b), the service interface is not modified but just its endpoint. However, we need to keep account of the service instances that are dynamically created or destroyed for each service in the Registry component of the CloudSocket prototype. As (b) is more appealing, then we follow it by also considering that the Cloud Provider Engine is able to dynamically modify the contents of the (service) registry.

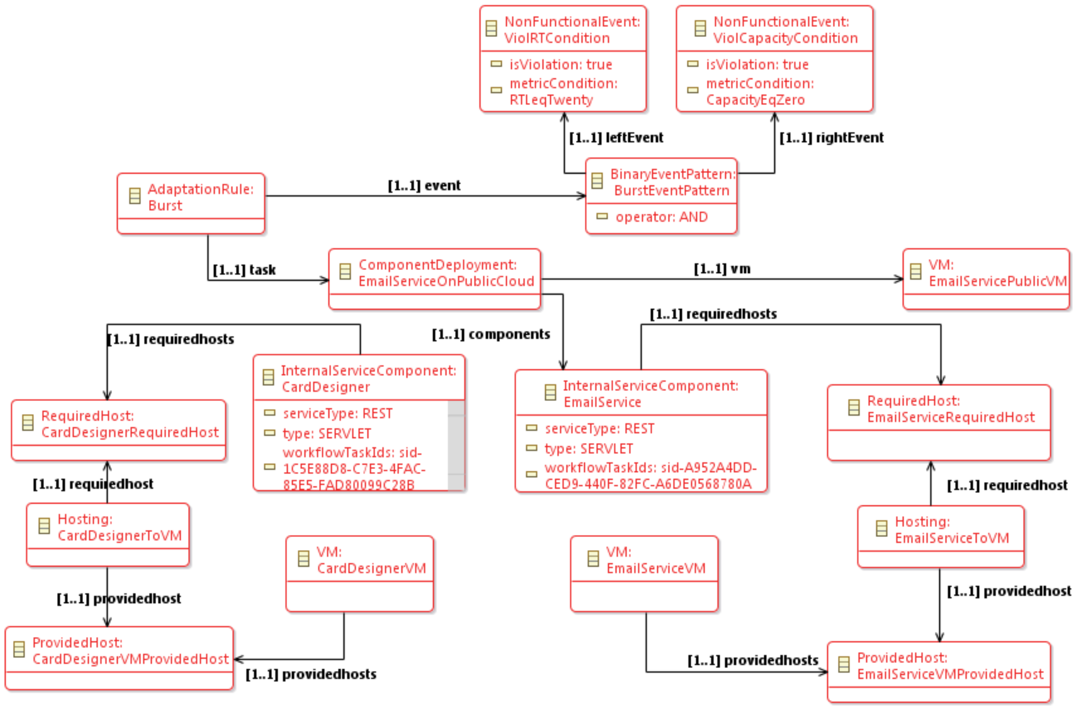

NonFunctionalEvent: ViolRTCondition
isViolation: true
metricCondition: RTLeqTwenty
NonFunctionalEvent: ViolCapacityCondition
isViolation: true
metricCondition: CapacityEqZero
[1..1] leftEvent
[1..1] rightEvent
AdaptationRule: Burst
[1..1] event
BinaryEventPattern: BurstEventPattern
operator: AND
[1..1] task
ComponentDeployment: EmailServiceOnPublicCloud
[1..1] vm
VM: EmailServicePublicVM
[1..1] requiredhosts
[1..1] components
[1..1] requiredhosts
InternalServiceComponent: CardDesigner
serviceType: REST
type: SERVLET
workflowTaskIds: sid-1C5E88D8-C7E3-4FAC-85E5-FAD80099C28B
InternalServiceComponent: EmailService
serviceType: REST
type: SERVLET
workflowTaskIds: sid-A952A4DD-CED9-440F-82FC-A6DE0568780A
RequiredHost: CardDesignerRequiredHost
RequiredHost: EmailServiceRequiredHost
[1..1] requiredhost
[1..1] requiredhost
Hosting: CardDesignerToVM
VM: CardDesignerVM
VM: EmailServiceVM
Hosting: EmailServiceToVM
[1..1] providedhost
[1..1] providedhost
ProvidedHost: CardDesignerVMProvidedHost
[1..1] providedhosts
[1..1] providedhosts
ProvidedHost: EmailServiceVMProvidedHost

Figure 9 - Fragment of CAMEL focusing on the description of the bursting scenario

Based on the above analysis, the respective adaptation rule to be specified in the CAMEL model will prescribe the following: If raw response time for email service component is greater than 0.5 minutes, then replace the current instance with a new one. This will involve specifying a *ServiceReplacement* object which will indicate that the *previousService* property value would map to the email sending internal service component while the *newService* attribute value will be empty to indicate the preference that the system should identify automatically the most appropriate replacement service. Please also note that the *taskIds* will also be empty, signifying that this change will have an effect over the whole BPaaS workflow instance and thus on all tasks being realised by the email service component instance.

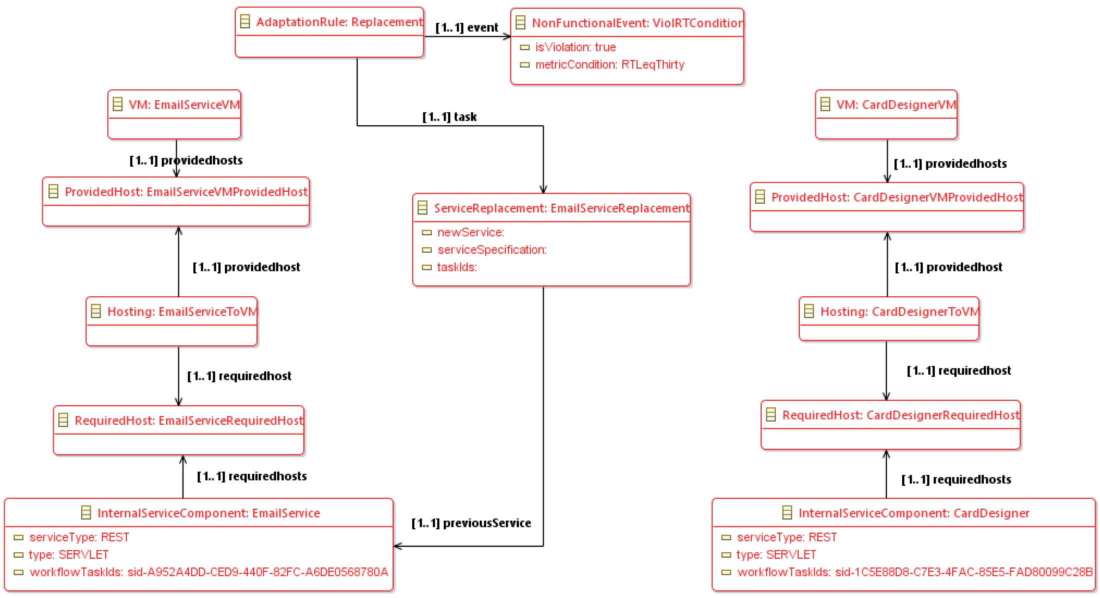

AdaptationRule: Replacement
[1..1] event
NonFunctionalEvent: ViolRTCondition
isViolation: true
metricCondition: RTLeqThirty
VM: EmailServiceVM
[1..1] task
VM: CardDesignerVM
[1..1] providedhosts
[1..1] providedhosts
ProvidedHost: EmailServiceVMProvidedHost
ServiceReplacement: EmailServiceReplacement
newService:
serviceSpecification:
taskIds:
ProvidedHost: CardDesignerVMProvidedHost
[1..1] providedhost
[1..1] providedhost
Hosting: EmailServiceToVM
Hosting: CardDesignerToVM
[1..1] requiredhost
[1..1] requiredhost
RequiredHost: EmailServiceRequiredHost
RequiredHost: CardDesignerRequiredHost
[1..1] requiredhosts
[1..1] requiredhosts
InternalServiceComponent: EmailService
serviceType: REST
type: SERVLET
workflowTaskIds: sid-A952A4DD-CED9-440F-82FC-A6DE0568780A
[1..1] previousService
InternalServiceComponent: CardDesigner
serviceType: REST
type: SERVLET
workflowTaskIds: sid-1C5E88D8-C7E3-4FAC-85E5-FAD80099C28B

Figure 10 - Fragment of CAMEL focusing on the description of the service replacement scenario

The respective part of CAMEL which is used to specify the adaptation rule is depicted in Figure 10, while the whole CAMEL model is provided in the appendix of this deliverable.

2.3.3 Future Work

2.3.3.1 PaaS Support

Currently, the SRL extension considers that the BPaaS has been mapped to a workflow which includes the exploitation of SaaS services and the usage of underlying IaaS resources to support its execution. To this end, the adaptation on the PaaS level is currently not considered. This is a rather rational choice as the respective PaaS extension in CAMEL has been developed in parallel with this extension. However, in the near future and possibly in the context of this project, we plan to further enhance this SRL extension to be able to address the missing level in the cloud computing stack.

2.3.3.2 Adaptation Task Coverage

The coverage of adaptation tasks has followed a pragmatic approach based on the actual coverage in the literature, especially with respect to corresponding proprietary frameworks and research prototypes. This corresponds to the most usual adaptation tasks at the IaaS level, the most frequently realised adaptation task at the SaaS level and well-known and supported adaptation tasks at the workflow level (useful in dynamic or critical situations (e.g., emergency) or when respective tasks are handled by humans (and can thus involve even delegation and splitting of a task functionality)). This pragmatic approach is assorted with respective adaptation capabilities that are or will be supported by the CloudSocket implementation. In the near future, when more advanced scenarios are to be addressed, then we do not preclude a further extension of SRL in order to cover their appropriate modelling and implementation of the corresponding extra adaptation capabilities.

3 ALLOCATION ENVIRONMENT PROTOTYPES

The prototypes of the BPaaS Allocation Environment will be used in the Allocation editor and KPI & SLA Editor (see Figure 11). The following prototypes *Smart Service Discovery and Composition* and *DMN to CAMEL Mapping* will be integrated in order to support the allocation decisions. The first approach allows an automatic allocation decision based on technical requirements, whereas the latter can be used by business consultants to have an increased usability while customizing an existing BPaaS bundle to specific use cases. Examples for this supportive behaviour are to help the system or user to find appropriate implementations for the workflow, or define rules to adapt a certain BPaaS bundle to the needs of a customer. In addition, these prototypes will ease the usage of the new features PaaS orchestration and cross layer service adaptation by reducing the complexity of the BPaaS bundle creation process.

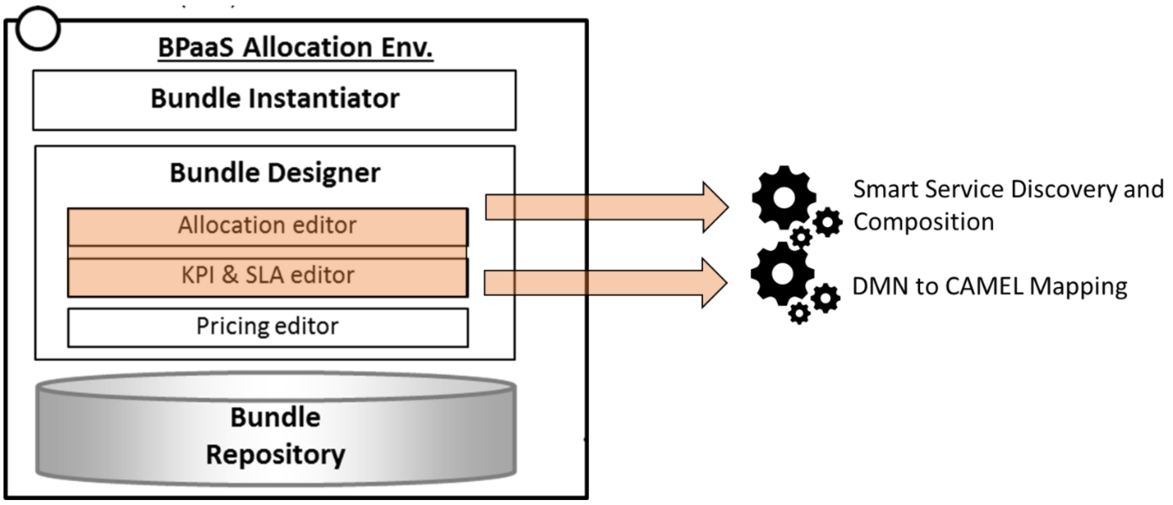


Figure 11 - Prototypes in the BPaaS Allocation Environment

3.1 Smart Service Discovery and Composition

The Smart Service Discovery and Composition blueprint enables both the discovery of cloud services at different levels of abstraction as well as the concurrent selection of both SaaS & IaaS services for service-based workflow concretisation according to the broker functional and non-functional requirements. The latter type of selection has been deemed as a necessity in order to support the discovery of a real optimised solution by considering that the selection at a lower level has an effect over the selection of a higher level at the cloud computing stack.

A detailed analysis of the blueprint can be found at Deliverable D3.3 [1]. In this section, we provide a summary of the blueprint, report its current status and specify important implementation and exploitation aspects for it while we unveil its short-term future extensions.

3.1.1 Features

The blueprint actually comprises two main components or sub-blueprints dedicated to cloud service discovery and selection, respectively. As such, the respective feature presentation is done per each main component of this blueprint.

Concerning service discovery, the main features of the blueprint can be summarised as follows:

- Supports both functional and non-functional semantic service discovery to cover all possible aspects in service description. The functional aspect is covered by OWL-S while the non-functional one by OWL-Q.

- Smart service discovery is offered which enables boosting the service matchmaking time. Smartness is realised at two levels: (a) the transactional combination of aspect-specific matchmakers according to different composition semantics (currently parallel composition maps to the fastest implementation); (b) at the individual, aspect-specific level, each service matchmaker utilises smart structures in order to better organise the service advertisement space and accelerate the service matchmaking.
- The blueprint enables not only the matchmaking but also the management (update, insertion, deletion) of the functional and non-functional service specifications.
- A Java and REST API are offered to enable the service discovery and service specification management.

We should highlight that the second feature was partially analysed in D3.3. The focus was then on the combination of different techniques for non-functional service matchmaking. In the meantime, to also support functional and thus combined service discovery, respective work has been conducted in order to explore the possible ways functional and non-functional service matchmaking algorithms can be combined in order to speed up the service discovery time. The main result of that work, where more details can be found in [5], were 4 main orchestration algorithms: (a) *sequential* with two versions focusing on first performing functional service matchmaking and then non-functional one. The main difference between these two versions is that one filters on the fly the services mapping to the functional results based on the user non-functional requirements while the second creates a respective repository out of the non-functional advertisements of these services and then applies a specific non-functional algorithm over them; (b) *parallel* where both functional and non-functional service matchmaking are performed in conjunction and then their common results, mapping to the same set of services, are joined. The main advantage of this algorithm is that it can stop whenever the fastest from the two aspect-specific algorithms returns no discovery result while its overall execution time is restrained only from the slowest of the two aspect-specific algorithms being exploited; (c) *subsumes* where a subsumes-based hierarchy between the service advertisements is created in order to speed up the matchmaking time in a similar fashion with respect to the non-functional service matchmaking implementation (see more details in D3.3), where now a combined matchmaking metrics is exploited taking into consideration both service description aspects; (d) *subsumedBy* where a similar hierarchy is constructed which now relies on the opposite to subsumes relation.

As far as service selection is concerned, a respective algorithm has been already developed (see D3.3 & [8]) which exhibits the following features:

- concurrent selection of both IaaS and SaaS services to deliver purely optimal selection solutions for the service-based BPaaS workflow.
- catering also for different realisation alternatives for a SaaS service: (a) either an external SaaS service can be exploited or (b) an internal one which also requires being hosted in a public or private cloud. In the latter case, the internal service code would be either purchased or has been developed internally by the broker organisation.
- consideration of a great variety of non-functional requirements, including performance and reliability ones (over, e.g., response time, throughput, availability and reliability), security ones at both a coarse (in terms of security controls) and a fine-grained level (in terms of security SLOs) and cost.
- capability to map low-level non-functional capabilities to higher-level ones in the form of dependency functions to cover any dependency gap within the respective optimisation problem.
- capability to handle both linear and non-linear constraints as well as both real and integer-based variables (mapping to the coverage of the previous functions as well as aggregation ones over non-functional metrics (from component/service to application/workflow level)).
- in case that a specific time deadline has to be provided, the respective solver can be configured to take it into consideration, thus being able to more rapidly produce solutions with a potential penalty over solution optimality.

- capability to handle over-constrained user/broker requirements via the use of flexible utility functions that enable a slight violation of user requirements in that case in order to always produce a meaningful solution to the broker.
- takes into account offerings which promise not just one but a range of values for each non-functional term considered - this is essential in order to be able to capture service level variability in highly dynamic environments.

Please note that the selection algorithm has been recently extended [6] to enable fixing parts of the problem according to the knowledge that is derived by a Knowledge Base by applying rules over the application execution history. While this feature is thus enabled, it cannot be immediately exploited in CloudSocket since the respective knowledge derivation functionality has not yet been realised in the context of T3.3 (see also D3.5 [7]). Once the latter realisation is done, then of course this feature will be exploitable, enabling the production of solutions in a much faster way.

3.1.2 Architecture

The architecture of the blueprint is an agglomeration of the architectures proposed in the respective publications in [5, 8] as well as in D3.3 [1]. This architecture, which is depicted in Figure 12, includes three main levels: UI, business logic and database while it certainly embraces service-orientation. It comprises the following components at the highest level:

- The *UI* component offers the respective visualisation and editing means to enable the initialisation of service discovery and workflow concretisation interactions. At the background, it performs REST calls over the next component, the REST *SDS* service.
- The REST *SDS* (*Service Discovery and Selection*) service encapsulates the whole functionality of the blueprint and enables its programmatic access via REST function calls. This functionality is offered via two different modules which focus on the two main parts of service discovery and selection.
- The *Service Discovery Module* enables performing functional and non-functional service discovery via the supply of a combined OWL-S and OWL-Q request.
- The *Service Selection Module* actually realises the service-based workflow concretisation functionality. As service discovery is a prerequisite for service-based workflow concretisation, this module is actually able to invoke the *Service Discovery Module* in order to obtain all possible (service) alternatives for each BPaaS workflow task.

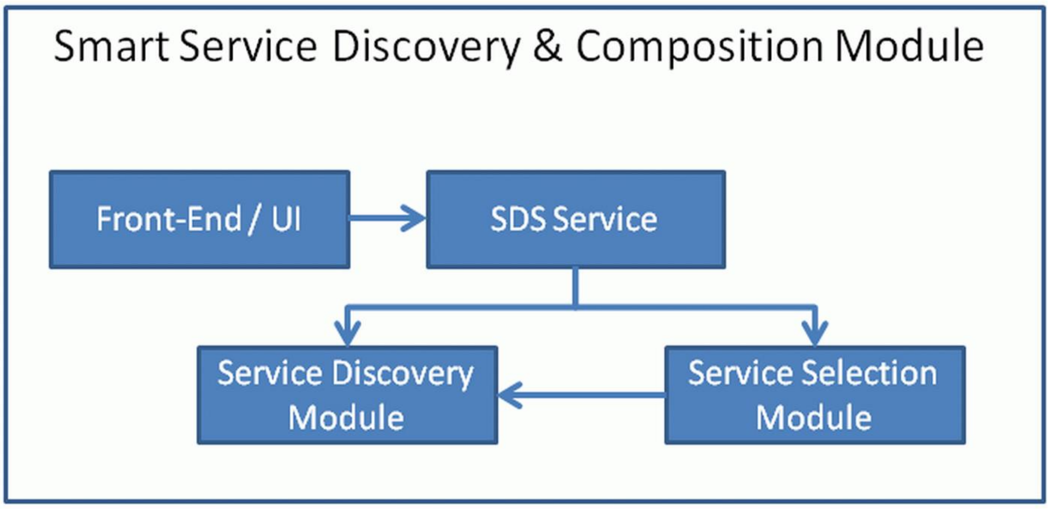


Figure 12 - The overall architecture of the Smart Service Discovery and Composition Module / Blueprint

This high-level architecture is further elaborated by concentrating on the internal architecture of the latter two modules. The latter architecture is also able to unveil the third level which was not apparent in the high-level architecture.

The architecture of the *Service Discovery Module* is depicted in Figure 13. As it can be seen, the *Specification Processor* is the component which takes the user/broker input and attempts to validate it both syntactically, semantically as well as in a constraint-based manner. Once user input is validated, its non-functional part is also aligned according to a specific non-functional alignment algorithm operating over a (non-functional) term store. Both the aligned non-functional specification and the functional one are passed to the *Compositor* which then issues them to the respective aspect-specific matchmakers, which are named as *Functional Matchmaking Module* and *NonFunctional Matchmaking Module*. Depending on the type of functionality requested, different actions are actually performed. In case of service registration / updating / deletion, the corresponding matchmaker performs the required update and then informs the *Compositor* for its main result. For service registration and deletion, transactionality is guaranteed by the latter component. This means that if for some reason, an aspect-specific registration / deletion fails, then the other aspect-specific registration is also rolled back to maintain a consistent state in the whole system. To assist in transactionality enforcement, a *Combined Registry* can be consulted and managed by the *Compositor* which enables storing the mappings between functional and non-functional specifications of services.

In case of service matchmaking, the *Compositor* realises the service discovery's composition logic by being able to execute different orchestration algorithms, as reported in D3.3 [1, 8], such as sequential and parallel ones. In any case, the facilities of the respective matchmaking modules are exploited in order to support the required aspect-specific discovery functionality. Once both aspect-specific discovery results are obtained, then these are relayed back from the *Compositor* to the *Specification Processor*.

The *Functional Matchmaking Module* is a slight modification of the service matchmaker that has been developed in the Alive European project [9]. Apart from being semantic, this matchmaker employs a smart structure and respective dynamic encoding scheme for domain ontologies that enables discovering concept ancestors or descendants in O(1) time, thus greatly speeding up the functional service matchmaking time where such ontology-based functions are highly needed and used. More about the internal details of this matchmaker can be found in [9]. For simplification reasons, the internal architecture of this module can be seen as a combination of two main components: (a) the functional matchmaking algorithm and (b) the Functional Store as the storage medium for the functional specification of services.

The *Non-Functional Matchmaking Module* is an agglomeration of different non-functional matchmaking algorithms which are able to exploit two different techniques to perform the matchmaking:

(a) constraint programming / solving and
(b) ontology subsumption.

All these algorithms have been extensively reported in D3.3 [1]. The respective internal architecture of this module was also reported in that deliverable. However, due to the integration with the combined service discovery work, this architecture was simplified as some of its components were moved out of this module at the level of the overall the *Service Discovery Module*. In this sense, the updated architecture of this module can be seen in Figure 13. As it can be seen, it comprises 6 main components. The *Broker* is responsible for orchestrating the appropriate actions towards conducting the service matchmaking. It actually realises internally the service matchmaking logic and exploits the facilities of other components in order to realise part of the needed functionality. In particular, in case of constraint-based service matchmaking, the respective service request needs to be transformed into a constraint (satisfaction) problem (CSP) via the *Transformer*. This CSP is then exploited in order to realise a corresponding matchmaking metric which also takes into account the CSP model of the corresponding non-functional service specification by exploiting the facilities of the *Constraint Solver*. In case of ontology-based service matchmaking, an *Ontology Reasoner* is exploited in order to infer the subsumption hierarchy between a pair of service request and offer or between the service request and all the non-functional offers, depending on the respective ontology-based non-functional service matchmaking algorithm to be exploited.

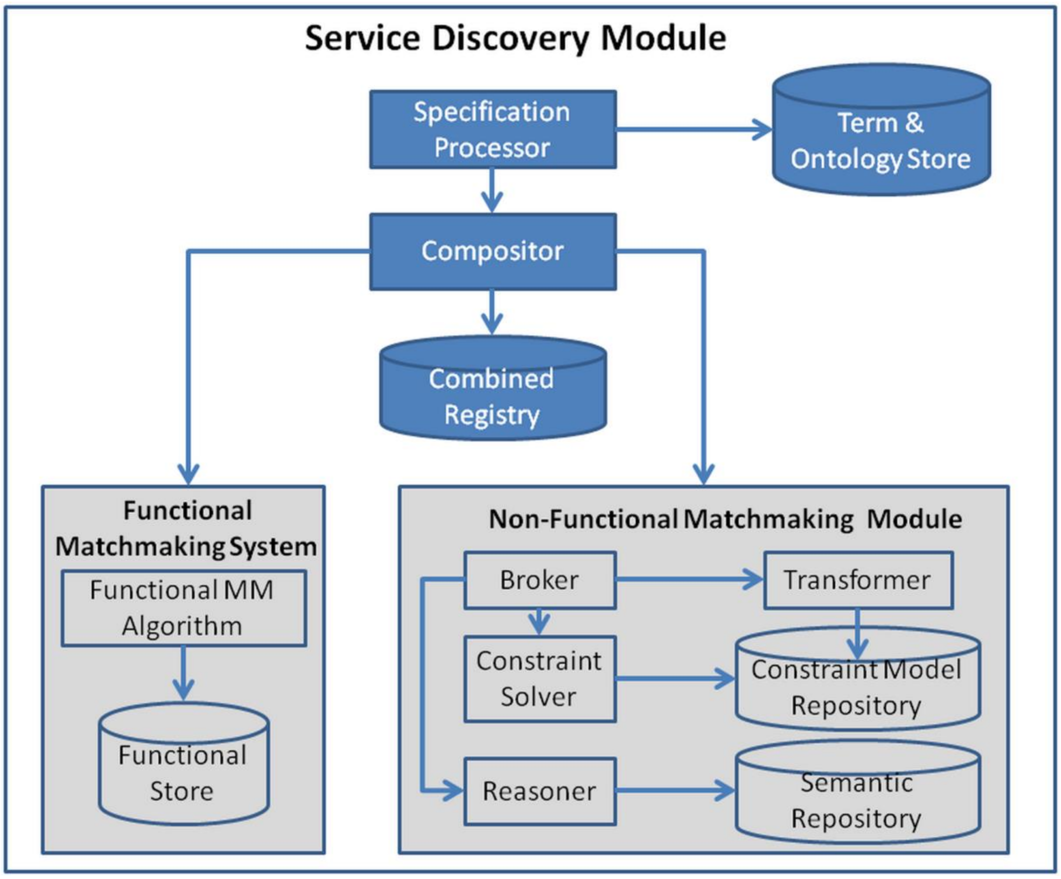


Figure 13 - The architecture of the Service Discovery Module

In case constraint-based service matchmaking is enabled via configuration, then obviously all the service offers should have been previously processed and transformed into CSP models which are stored in a *Constraint Model Repository*. On the other hand, in case that ontology-based service matchmaking is enabled, the respective service offers are stored in an internal *Semantic Repository*.

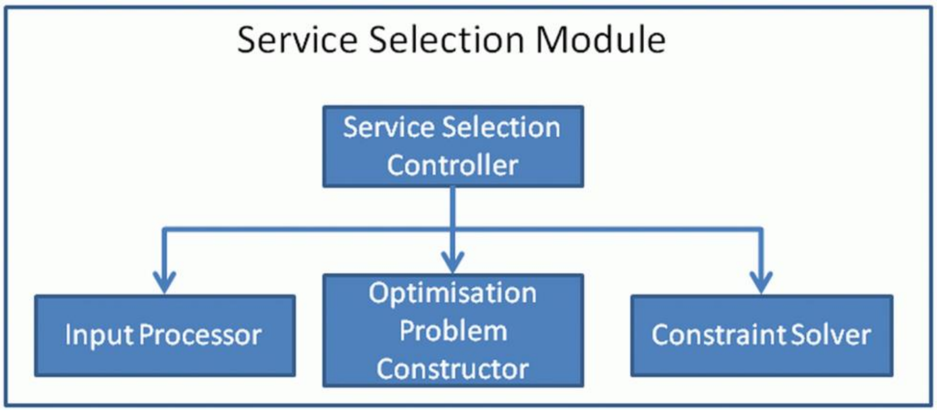

Service Selection Module
Service Selection Controller
Input Processor
Optimisation Problem Constructor
Constraint Solver

Figure 14 - The architecture of the Service Selection Module

The *Service Selection Module* follows a simplified architecture, depicted in Figure 14, which comprises three main components: (a) the *Service Selection Controller* is responsible for the orchestration of the components involved in the service-based workflow concretisation. This component first invokes the *Input Processor* in order to construct the appropriate input for the actual optimisation problem construction. This means that based on the respective semantically-annotated service-based BPaaS workflow, the *Input Processor* performs service discovery for all the tasks involved according to both task-based functional and non-functional broker requirements provided in the form of annotations. This also means that this component takes into account the BPaaS CAMEL model in order to discover those IaaS offerings which satisfy the respective requirements posed for the BPaaS internal service components. Once all input has been gathered and extended, the *Service Selection Controller* invokes the *Optimisation Problem Constructor* with it in order to transform it into a constraint optimisation problem. The later problem is finally sent by the *Service Selection Controller* to the *Constraint Solver* for solving.

3.1.3 Setup

The whole blueprint can be found as a Java project in the UULM's git repository[6]. As it is a maven project, it can be compiled and run via issuing well-known maven commands. The core functionality is offered as a REST service. Thus, the final compilation result is a war file which can be then deployed on a servlet container like tomcat. The UI component has not been yet developed. It is speculated whether it will be a new component or rely on the respective UI component of the *Allocation Environment*. Full documentation of this component can be found in the project wiki[7].

3.1.4 Future work

3.1.4.1 Service Composition

In cases where there is no SaaS able to realise the functionality of a particular BPaaS workflow task, then there is a high need to implement service composition functionality. The latter implementation can rely on our previous work on semantic service planning [10] which however relies on the existence of rich service specifications. An alternative would be to either use this semantic planning work based only on I/O or, if this is not supported, adopt a semantic I/O based planner. In any case, we need to check here the following: (a) whether there is a real scenario for which service composition is needed in the context of the CloudSocket use cases; (b) in case that such a scenario exists, inspect the actual way service composition functionality could be exploited. In the latter case, the most natural way would be to obtain all possible abstract service plans able to realise the required BPaaS workflow task functionality and select the best possible one according to service plan criteria. In this way, the BPaaS workflow will be actually extended to include new tasks for which service selection will then need to be applied.

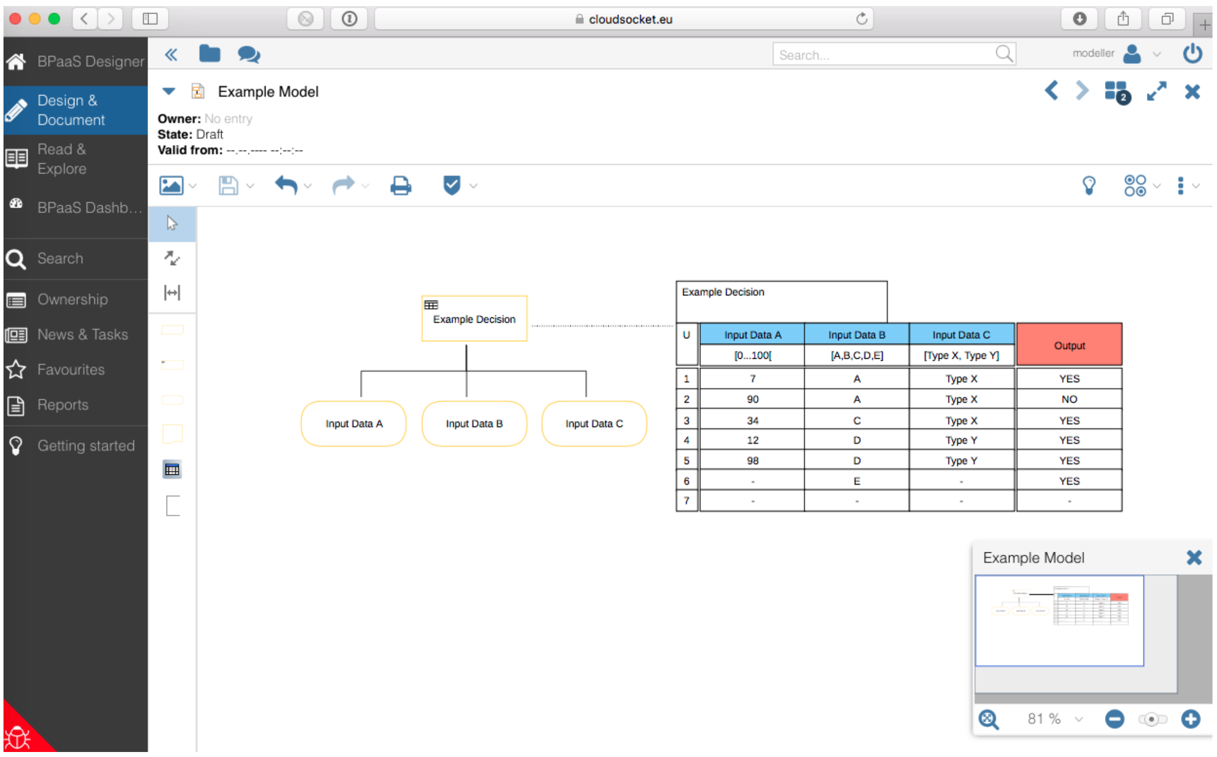
3.2 DMN to CAMEL Mapping

The DMN-to-CAMEL-Mapper prototype [1, 14] reduces the technical complexity of the software component allocation by mapping high-level business requirements to the low-level cloud-specific description. As business experts still require technical assistance for consuming cloud services and allocating the software components, the DMN-to-CAMEL-Mapper aims to create a way to semi-automatically handle the software component allocation and configuration based on high-level parameters. In this respect the Decision Model and Notation (DMN) [13] is applied. DMN is an industry standard for modeling and executing decisions that are determined by business rules. The DMN standard provides a human-readable common notation for modelling and automating decisions. We choose decision tables (DTs) to represent decisions as these are well known to business experts.

[6] https://omi-gitlab.e-technik.uni-ulm.de/cloudsocket/service_discovery_composition

[7] https://www.cloudsocket.eu/common-understanding-wiki/-/wiki/Main/Smart+Service+Discovery+and+Composition+Blueprint

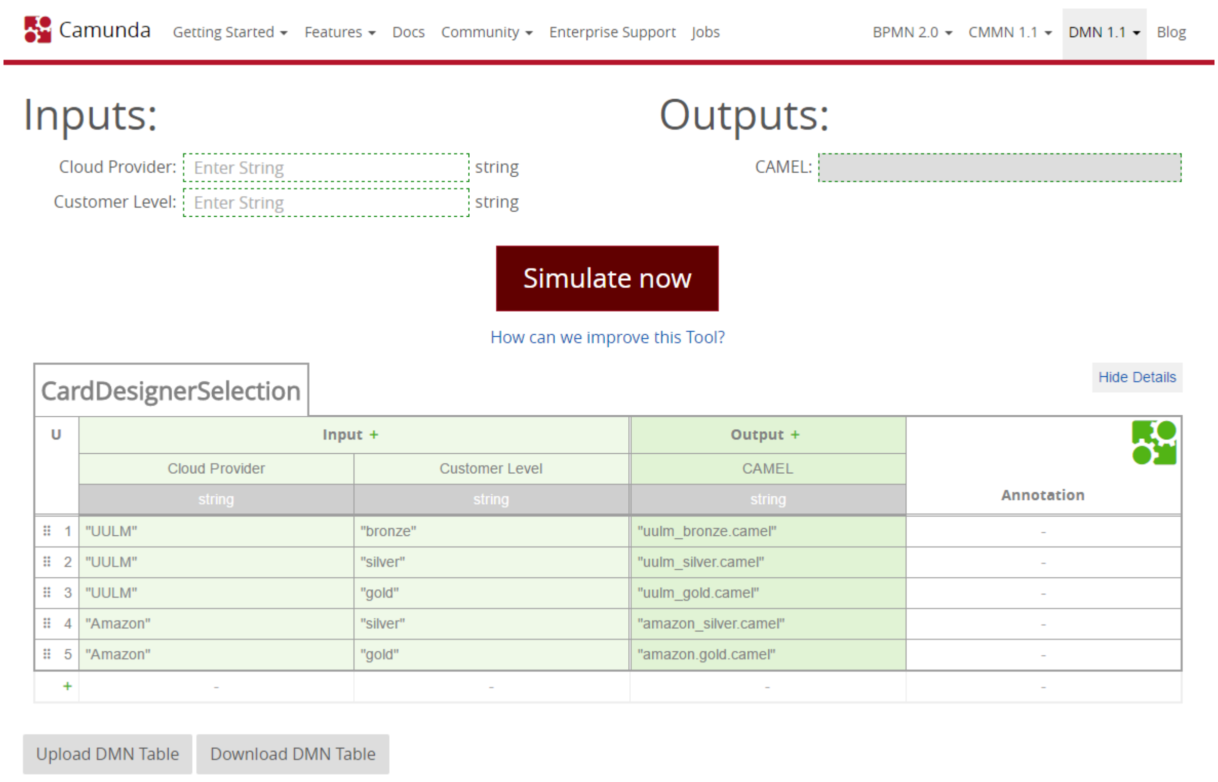
This approach enables the modelling of cloud applications by using non-technical business values relating to business requirements, which will be mapped to the technical CAMEL model.



Owner: No entry
State: Draft
Valid from: --.--.---- --:--:--

Example Decision

U	Input Data A	Input Data B	Input Data C	Output
	[0...100[	[A,B,C,D,E]	[Type X, Type Y]	
1	7	A	Type X	YES
2	90	A	Type X	NO
3	34	C	Type X	YES
4	12	D	Type Y	YES
5	98	D	Type Y	YES
6	-	E	-	YES
7	-	-	-	-



Inputs:

Cloud Provider: Enter String string

Customer Level: Enter String string

Outputs:

CAMEL:

CardDesignerSelection

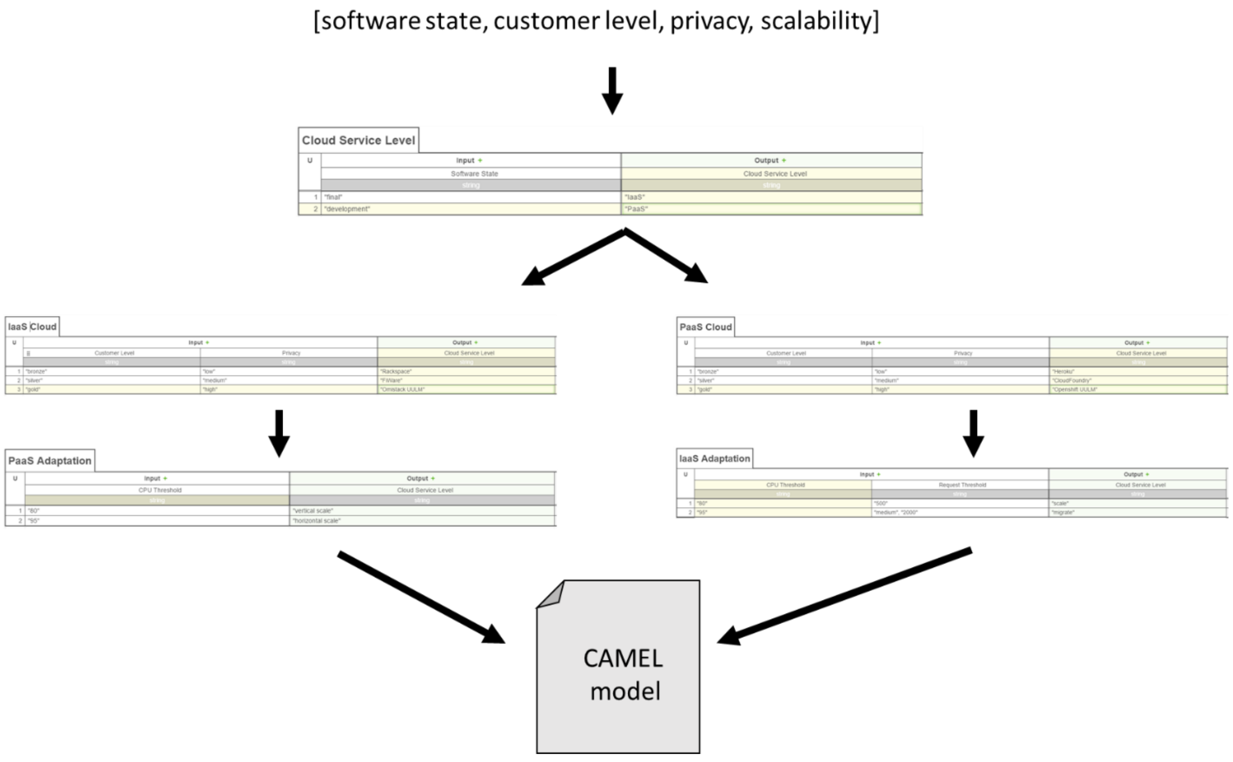
U	Input +		Output +	
	Cloud Provider	Customer Level	CAMEL	Annotation
	string	string	string	
1	"UULM"	"bronze"	"uulm_bronze.camel"	-
2	"UULM"	"silver"	"uulm_silver.camel"	-
3	"UULM"	"gold"	"uulm_gold.camel"	-
4	"Amazon"	"silver"	"amazon_silver.camel"	-
5	"Amazon"	"gold"	"amazon.gold.camel"	-
+	-	-	-	-

Figure 15 - DMN modelling tools: ADOxx (up) and Camunda Editor (down)

In terms of the running example, the DMN-to-CAMEL-Mapper can be used for creating different kind of deployment models for the ChristmasCardDesigner service. An example scenario, which can be solved with the prototype, is the following: The CAMEL deployment model fragment should be generated based on the desired cloud provider and the customer level *bronze, silver* and *gold*, where the business values are mapped to the respective virtual machine flavours. Therefore, the required DMN tables need to defined, which can be edited via the web-based editors offered by ADOxx[8] or Camunda[9], as depicted in Figure 15.

Defined DMN tables are registered at the DMN-to-CAMEL-Mapper and are executed via the provided REST API. The result of an exemplary call to get the deployment CAMEL fragment for the parameters cloud `provider = UULM` and customer `level = gold` can be found in the Annex.

A more complex example with connected DMN tables could be used to determine the placement of the ChristmasCardDesigner software component based on the following business values: `software state`, `customer level`, `privacy`, and `scalability`. The connected tables are shown in Figure 16. A first table can determine if the software component should be deployed on IaaS if the `software state` is final while PaaS is selected when the software state is `development` as PaaS supports the easy integration of continuous delivery of new software versions. Based on the decision connected tables can select the actual cloud provider, either a IaaS or a PaaS provider, based on the `customer level` and `privacy` values. Finally, is also possible to derive adaptation actions based on the `scalability` values, e.g. expected amount of users. In the end, a CAMEL model is returned, comprising the model fragment for the selected cloud provider and a model fragment with adaptation actions.



Cloud Service Level

U	Input +	Output +
	Software State	Cloud Service Level
	string	string
1	"final"	"IaaS"
2	"development"	"PaaS"

IaaS Cloud

U	Input +		Output +
	Customer Level	Privacy	Cloud Service Level
	string	string	string
1	"bronze"	"low"	"Rackspace"
2	"silver"	"medium"	"FiWare"
3	"gold"	"high"	"Omistack UULM"

PaaS Cloud

U	Input +		Output +
	Customer Level	Privacy	Cloud Service Level
	string	string	string
1	"bronze"	"low"	"Heroku"
2	"silver"	"medium"	"CloudFoundry"
3	"gold"	"high"	"Openshift UULM"

PaaS Adaptation

U	Input +	Output +
	CPU Threshold	Cloud Service Level
	string	string
1	"80"	"vertical scale"
2	"95"	"horizontal scale"

IaaS Adaptation

U	Input +		Output +
	CPU Threshold	Request Threshold	Cloud Service Level
	string	string	string
1	"80"	"500"	"scale"
2	"95"	"medium", "2000"	"migrate"

Figure 16 - Connected DMN tables for ChristmasCardDesginer deployment

3.2.1 Features

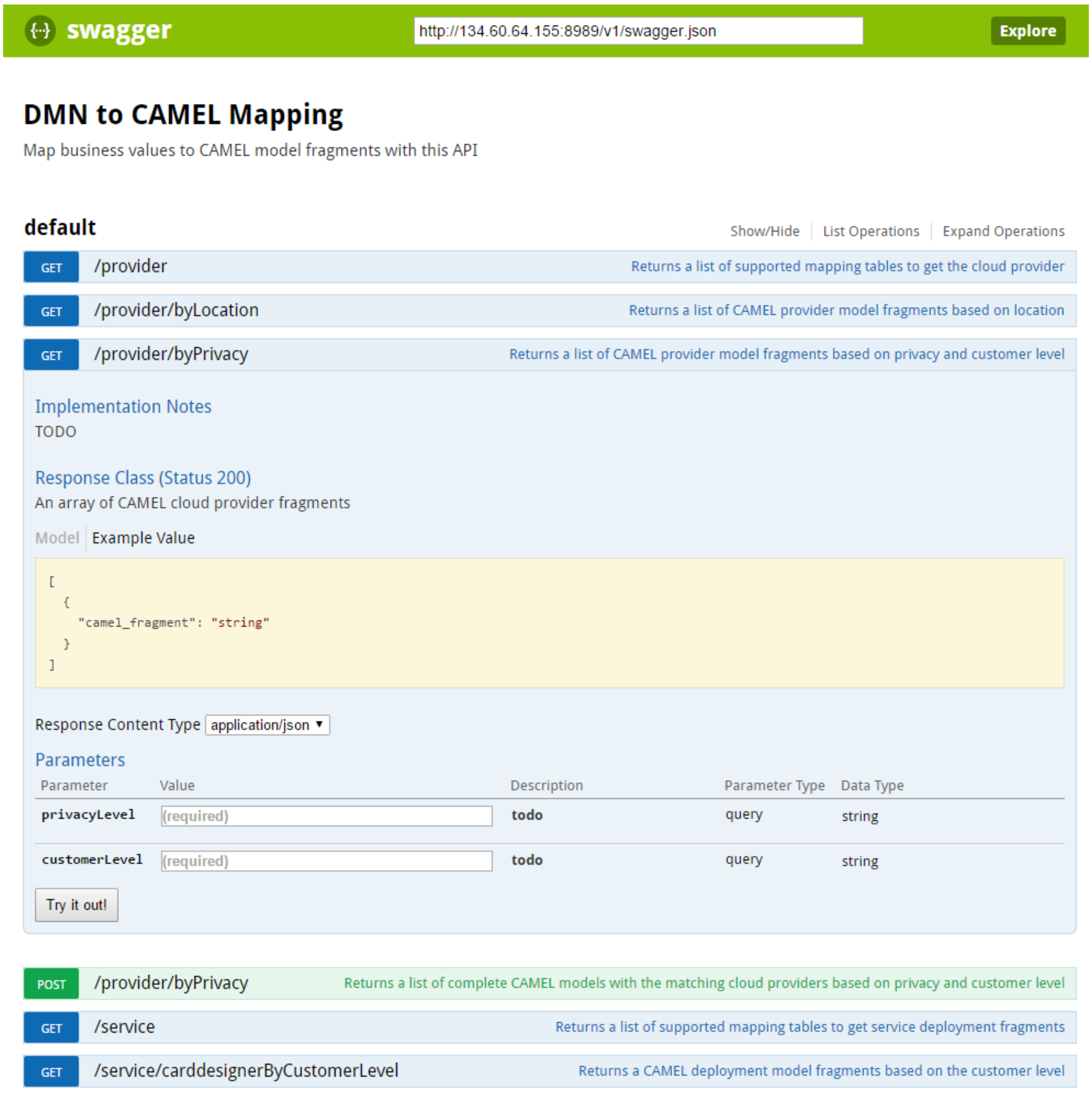
- Executing DMN 1.1 tables to generate CAMEL model fragments
- Mapping high level business values to technical CAMEL fragments

[8] https://www.cloudsocket.eu/BPAASDesigner/login.view
[9] https://camunda.org/dmn/simulator/

- Executing of multiple interconnected DMN tables, enabling hierarchical mapping of business values down to technical model fragments
- Current support for deployment and cloud provider modelling fragments
- Modular and easy extensible architecture
- Access via an easy to use REST API or web-based UI

3.2.2 Architecture

A high-level overview of the DMN-to-CAMEL-Mapper architecture is provided in Figure 18. To execute a mapping, the prototype provides a REST interface as well as a web-based UI (see Figure 17). Due to the use of the Swagger REST API description, it is possible to auto-generate also clients for all common programming languages that eases the integration in existing systems.



DMN to CAMEL Mapping

Map business values to CAMEL model fragments with this API

default — Show/Hide | List Operations | Expand Operations

GET /provider — Returns a list of supported mapping tables to get the cloud provider

GET /provider/byLocation — Returns a list of CAMEL provider model fragments based on location

GET /provider/byPrivacy — Returns a list of CAMEL provider model fragments based on privacy and customer level

Implementation Notes

TODO

Response Class (Status 200)

An array of CAMEL cloud provider fragments

Model | Example Value

```
[
  {
    "camel_fragment": "string"
  }
]
```

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
privacyLevel	(required)	todo	query	string
customerLevel	(required)	todo	query	string

Try it out!

POST /provider/byPrivacy — Returns a list of complete CAMEL models with the matching cloud providers based on privacy and customer level

GET /service — Returns a list of supported mapping tables to get service deployment fragments

GET /service/carddesignerByCustomerLevel — Returns a CAMEL deployment model fragments based on the customer level

Figure 17 - DMN to CAMEL Mapping web interface

Mapping calls are internally processed by translating these calls to the respective DMN tables in the DMN table repository. DMN tables need to be modelled beforehand and stored in the DMN model repository. The DMN-to-CAMEL-Mapper supports all DMN tables modelled in DMN version 1.1.

The output parameters of DMN tables can link to two options: *(1)* output values and another DMN table to execute, which allows a hierarchical mapping of multiple DMN tables *(2)* a CAMEL fragment from the CAMEL fragment repository in XMI or textual format which will be returned by the DMN-to-CAMEL-Mapper.

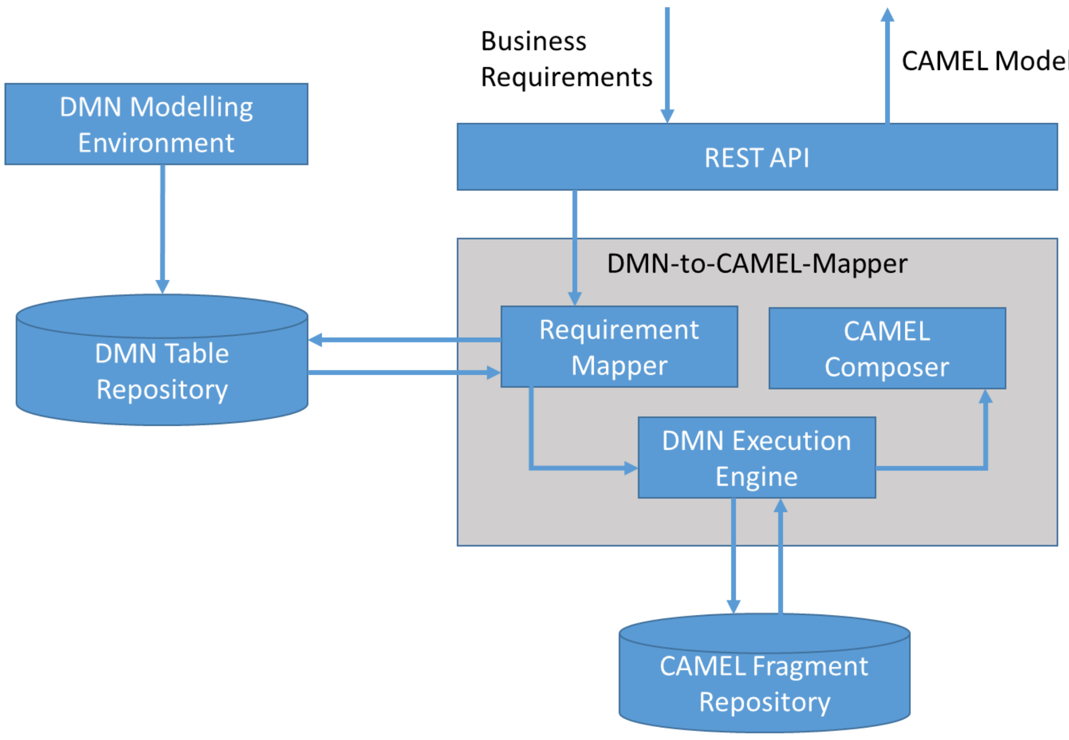


Figure 18 - DMN-to-CAMEL-Mapper high level architecture

3.2.3 Setup

The prototype is a standalone server application that has no external dependencies to other components. The sources of the prototype are publicly available on the GitLab of UULM[10]. A setup and usage guide can be found in the CloudSocket Wiki[11].

A running instance is available at http://134.60.64.155:8989/index.html and will later be referenced on the CloudSocket webpage.

3.2.4 Future work

The extensible architecture of the prototype offers the following extension points for additional features that can be followed in the context of WP4

- Support for mapping business values to CAMEL adaptation model fragments
- Update existing CAMEL models based on updated business values
- Composition of multiple DMN tables mapping to create complete CAMEL models

[10] https://omi-gitlab.e-technik.uni-ulm.de/cloudsocket/dmn-to-camel-mapper
[11] https://www.cloudsocket.eu/common-understanding-wiki/-/wiki/Main/DMN-to-CAMEL+Mapper

4 EXECUTION ENVIRONMENT PROTOTYPES

The prototypes of the BPaaS Execution Environment mainly circle around the Adaptation Engine, Monitoring Engine and the Cloud Provider Engine (see Figure 19). For the latter engine, we introduce capabilities to manage Cloud applications also on the PaaS layer. For the Monitoring and Adaptation Engines, we developed enhanced synergic frameworks. These also feed the SLA Engine partly in terms of capturing and ensuring more fine-grained SLOs.

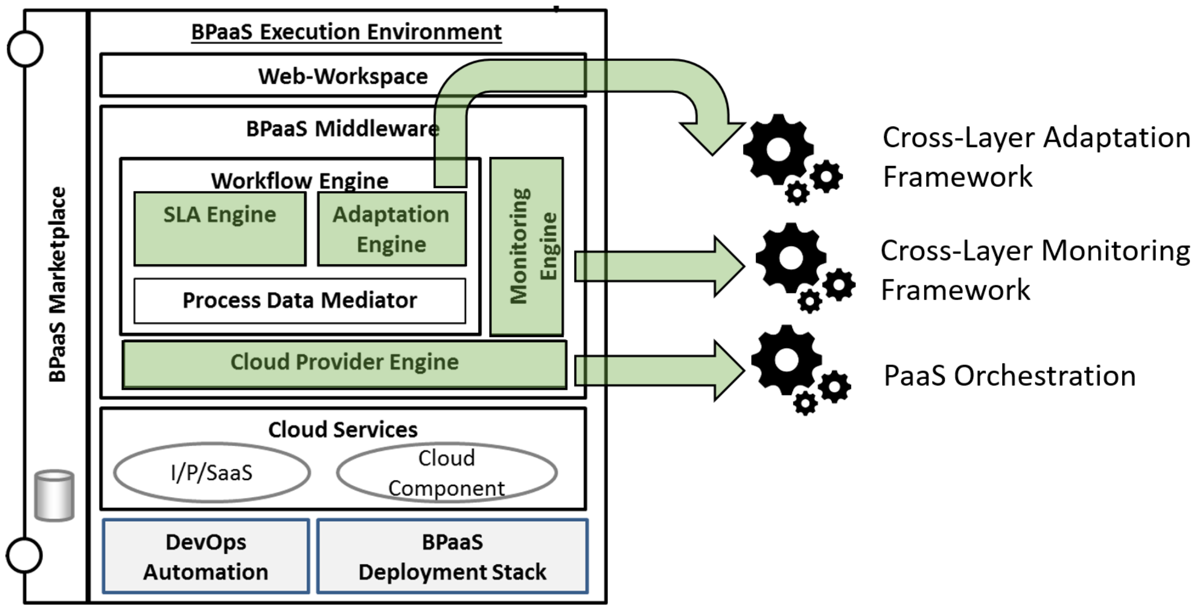


Figure 19 - Main affected components in the BPaaS Execution Environment.

4.1 PaaS Orchestration

The Cloud Provider Engine of CloudSocket, namely the Cloudiator Framework, has focussed on the orchestration of multi-cloud deployments on the IaaS level. In order to enable the usage of additional cloud service levels the Cloud Provider Engine is extended by the PaaS Orchestration Prototype to support the orchestration on the PaaS level based on the concepts described in Deliverable D3.3 [1]. While the multi-cloud orchestration on the IaaS level has already been addressed by different research projects[12], the multi-cloud orchestration on the PaaS level is a quite new research problem, which includes the modelling of PaaS applications (cf. PaaS extension of Camel in Section 2.1), the orchestration of PaaS applications and an abstraction layer over PaaS providers. Hence, the PaaS Orchestration Prototype comprises of the following prototype extensions to the Cloudiator components Shield and Colosseum [3] as well as a new component, called the PaaS-Unified-Library (PUL). These 3 components are analysed in the following three sub-sections.

4.1.1 Colosseum

In order to allow the definition of PaaS-based components and their deployment, we extended the model of Cloudiator by the entities shown in Figure 20.

[12] For example PaaSage (http://www.paasage.eu/) and Cactos (http://www.cactosfp7.eu/)

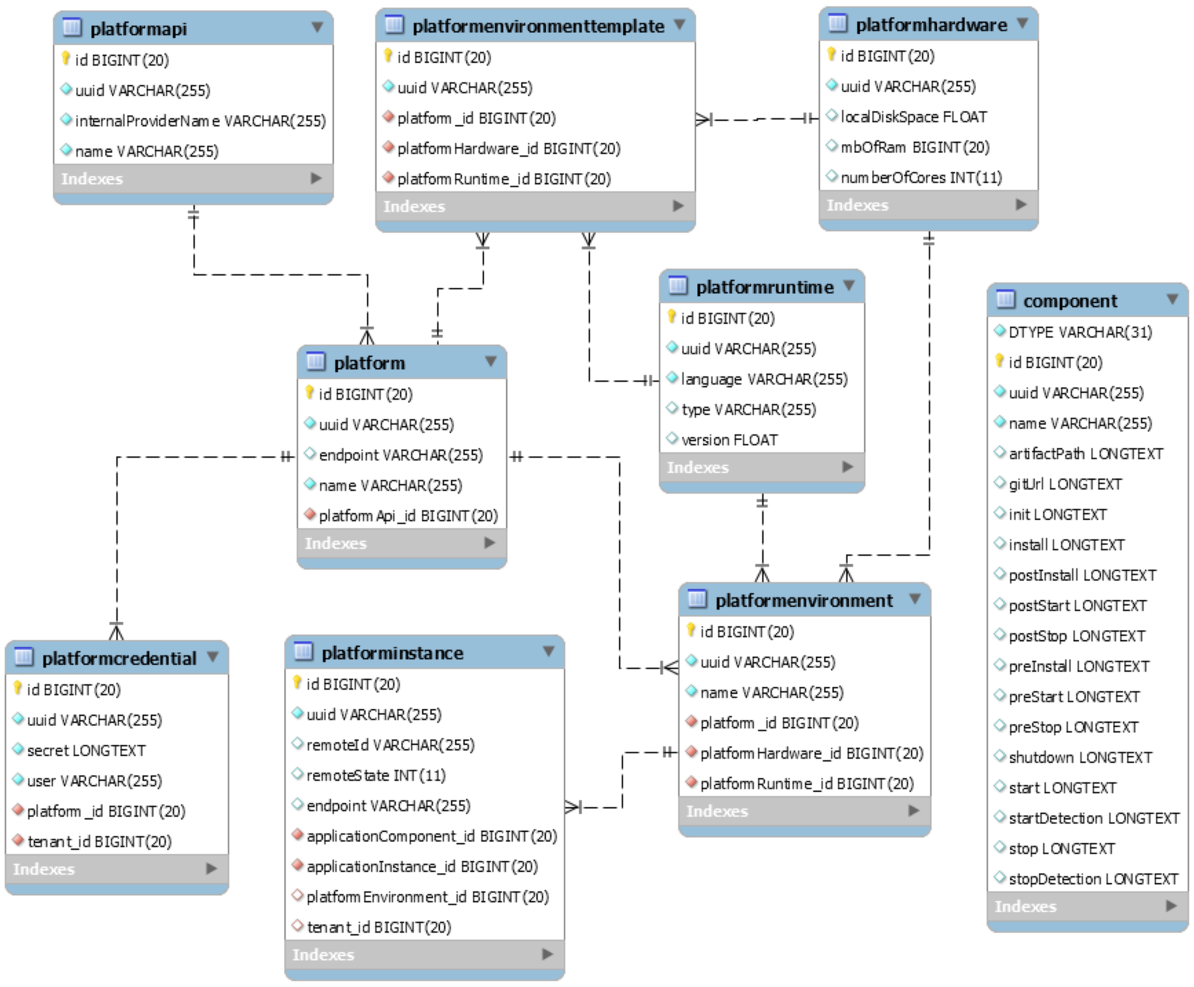


Figure 20 - Database scheme of Cloudiator for the PaaS integration.

Platform holds Provider-specific data like endpoint, API description and credentials. This is for the system to be able to connect and authenticate to a specific provider.

PlatformApi describes the actual API that is later used to decide how to communicate with a certain PaaS provider. A *PlatformComponent* describes the deployment and configuration of a Component on the PaaS layer. In Figure 20 it is part of the class Component, as an extension to the available properties. This is currently mainly defined by the API of the PaaS Unified Library server, i.e. mostly the artifact or a git URI that has to be provided to deploy a component. *PlatformCredential* holds the user authorization data, such as user name and a secret.

PlatformEnvironment defines the needed *PlatformRuntime* and *PlatformService* for a given *PlatformComponent*. A *PlatformEnvironmentTemplate* abstracts from the former *PlatformEnvironment* in the form of allowing to define an initial template that is used in case of horizontal scaling or migration, as it describes the initial template of a *PlatformEnvironment*, which can change over time (e.g. due to vertical scaling).

PlatformHardware holds information about the hardware-specification of the *PlatformEnvironment*, such as available memory and provisioned CPUs. *PlatformInstance* is the actual running instance on the PaaS provider and is started, stopped and deleted by Cloudiator via the manipulation of this entity.

The *PlatformMonitor* enables the integration of application- and PaaS-platform-specific monitoring data into the monitoring system of Cloudiator. You can specify the components and instances to be monitored and define sensors for them. For this we run a Visor instance with the needed sensors on the same machine as Cloudiator that collects (push and pull) monitoring data from the PaaS provider towards a certain instance.

PlatformRuntime defines the software resources that are needed by the component and which will be installed into the *PlatformEnvironment* on which the Instance will be running.

Asynchronous background jobs perform the creation of a *PlatformInstance* equivalently to the creation of the deployment of components on IaaS-level. We currently directly access the PaaS Unified Library, developed within this project, via REST calls but we aim at integrating it later on in our own provider-abstraction layer called Sword.

The new research model entities can be found on github[13]. Along with their specific APIs, we have introduced them into the Colosseum-Client[14] Finally, we have created a simple sample application that later can be used to understand the way to define a PaaS-based application[15].

4.1.1.1 Features

In order to enable the PaaS orchestration, Colosseum has been extended with the following features:

- Interfaces to store and update PaaS specific environment parameters. With respect to the Card Designer Service these parameters are programming `language=JAVA, type=Servlet, version=1.7`
- Interfaces to store PaaS resource parameters such as `memory` or `cpu cores`
- Interfaces to bind additional PaaS services to the actual applications, e.g. bind a MySQL service offered by the PaaS provider to the actual PaaS service, e.g. the Card Designer Service.
- Orchestration interfaces for PaaS applications, providing the following actions: create, start, delete, update, and scale.

4.1.1.2 Architecture

As the Colosseum component of Cloudiator is the main component for orchestration, several features have been implemented to enable the PaaS orchestration Figure 21 shows the original internal architecture of Colosseum where the green shapes show extended internal components, orange shapes show new components and blue shapes show the new PaaS orchestration feature. The latter just shows some of the available features provided by respective PaaS services.

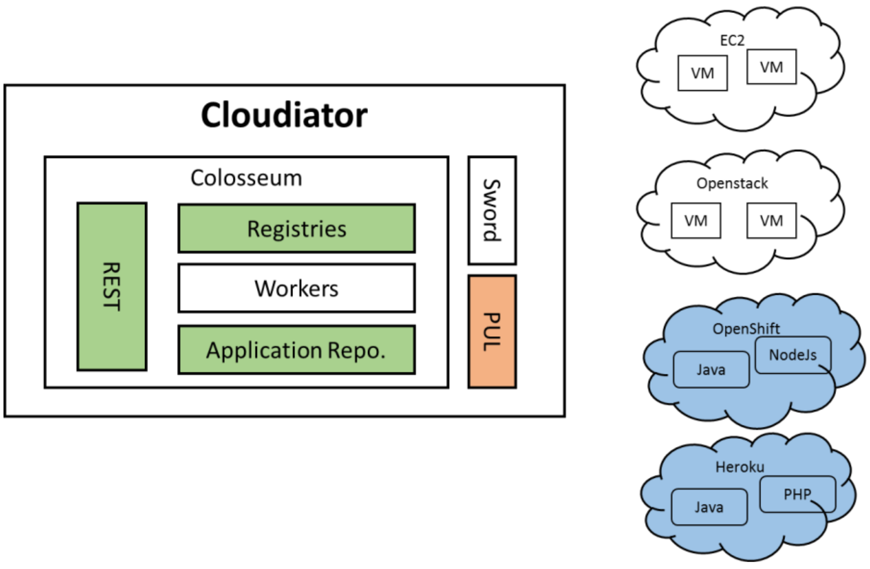


Figure 21 – High-level Cloudiator architecture.

[13] https://github.com/cloudiator/colosseum/tree/cs-paas-prototype/app/models
[14] https://github.com/cloudiator/colosseum-client/tree/cs-paas-prototype
[15] https://omi-gitlab.e-technik.uni-ulm.de/cloudsocket/cpe-paas-example

The REST interface has been extended in order to provide all the listed features of the previous section. The registries now store not only IaaS specific resources, but also PaaS specific resources. The application repository contains now IaaS and PaaS applications, which enables the composition of complex services, partially running on IaaS and PaaS providers.

4.1.1.3 Setup

As Colosseum is part of the Cloud Provider Engine, which requires to be a stable component in the CloudSocket execution workflow, a separate branch for the PaaS orchestration prototype has been created[16]. This branch is integrated according to the integration process [T4.5] which will be described in the upcoming deliverable "D4.9 CloudSocket Integration Report" and requires the respective branch of the Shield components to be operational. A technical setup description along with the complete documentation can be found in the CloudSocket Wiki[17].

4.1.1.4 Future work

4.1.1.4.1 Integration into Existing Abstraction Layer

Currently, the PaaS Unified Library server is running outside the Cloudiator framework and is used to communicate with the PaaS provider via own background jobs in Colosseum itself. We plan to more closely integrate the PaaS management into the abstraction layer Sword of Cloudiator. This will enable us to integrate also other PaaS management libraries as described in D3.3 [1]. This is similar to introducing cloud abstraction libraries, such as Apache jclouds[18], into Sword, as we did for the IaaS layer in order to re-use existing tools, but extend them by needed features.

4.1.1.4.2 Discovery of PaaS capabilities

Colosseum is able to automatically discover the capabilities of IaaS providers. This is later used to find optimal providers and deployment strategies. This feature will also be covered for PaaS providers in the future.

4.1.2 Shield

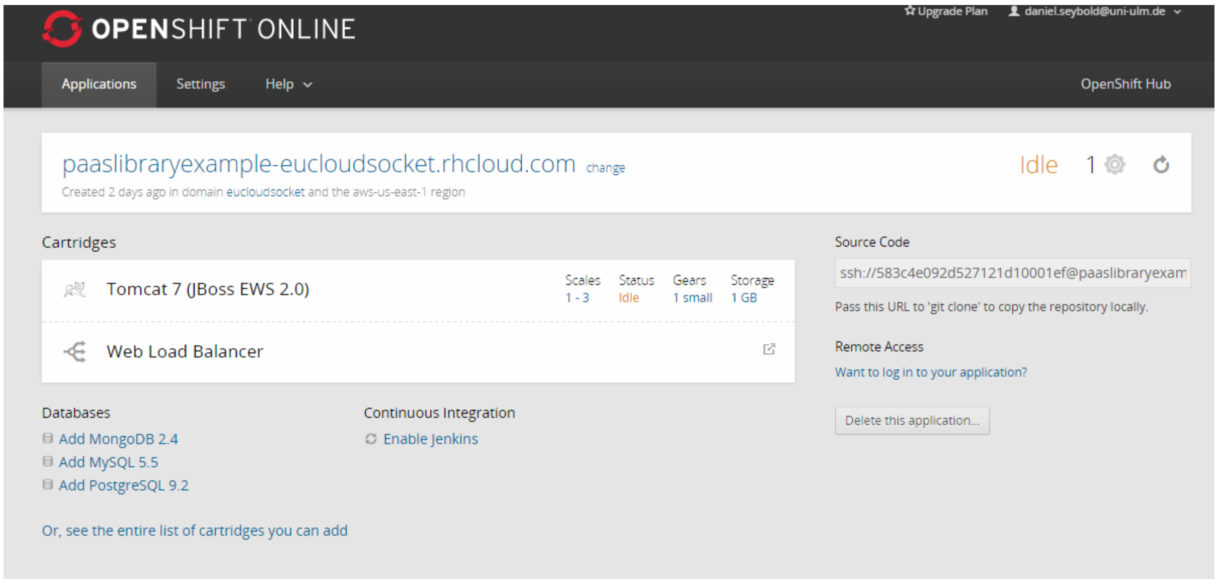
Shield is the adapter component of Cloudiator that currently implements the adaptation from CAMEL models to deployments in the Cloudiator framework. As CAMEL has run through greater updates when it comes to deployment description and adaptation plans, this CAMEL adapter had to be reorganised to fit the new needs. The current *research* version can be found here[19].

4.1.3 PaaS-Unified-Library (PUL)

In D3.3 [3], two tools were proposed to be integrated in the PaaS abstraction layer: PaaS Unified Library and COAPS API. The PaaS Unified Library was finally selected due to its maintainability and the on-going support by one of the partner in the consortium (i.e. ATOS) that will lead to improved bug-fixing and development of new features. As introduced in D3.3, the PaaS Unified Library is an outcome of the SeaClouds EU project[20], and it is being improved in CloudSocket. It comprises

(a) a library that provides simple operations for managing applications in PaaS providers,
(b) a REST interface on top of it, allowing the use of the library as a standalone application, and
(c) a Java client.

[16] https://github.com/cloudiator/colosseum/tree/cs-paas-prototype
[17] https://www.cloudsocket.eu/common-understanding-wiki/-/wiki/Main/Cloud+Provider+Engine+Component
[18] https://jclouds.apache.org/
[19] https://omi-gitlab.e-technik.uni-ulm.de/cloudsocket/camel-adapter/tree/cs-paas-prototype
[20] http://www.seaclouds-project.eu/



paaslibraryexample-eucloudsocket.rhcloud.com change
Created 2 days ago in domain eucloudsocket and the aws-us-east-1 region

Idle 1

Cartridges

	Scales	Status	Gears	Storage
Tomcat 7 (JBoss EWS 2.0)	1 - 3	Idle	1 small	1 GB
Web Load Balancer				

Databases
Add MongoDB 2.4
Add MySQL 5.5
Add PostgreSQL 9.2

Continuous Integration
Enable Jenkins

Or, see the entire list of cartridges you can add

Source Code
ssh://583c4e092d527121d10001ef@paaslibraryexam
Pass this URL to 'git clone' to copy the repository locally.

Remote Access
Want to log in to your application?

Delete this application...

Figure 22 – Service that was deployed on OpenShift Online via PaaS Unified Library

4.1.3.1 Features

This is the list of original features of the PaaS Unified Library:

- Unified API for managing PaaS providers
- The supported providers are Heroku, OpenShift v2 and CloudFoundry v2
- Operations: deploy, undeploy, start, stop, scale and bind service

The list of improvements implemented in CloudSocket is:

- Java client
- Different handling of credential headers.
- Initial Provider-independent deployment model
- Ability to support new providers by just adding runtime dependencies, with no need to recompile the whole library. This also allows to deploy PUL instances that support not all PaaS provider, but some of them. See the PUL architecture section below.
- API now includes ApiVersion as a parameter. A provider with two different API versions (e.g. OpenShift v2 and v3) were previously modelled as two different providers (mapped for example, in /openshift2 and /openshift3). Now, both versions are mapped in /openshift, and a parameter selects the version to use.

The Cloud Provider Engine uses the PaaS Unified Library to communicate with different PaaS providers in a standardised uniform way.

4.1.3.2 Architecture

The basic architecture of a server running the PaaS Unified Library is shown in Figure 23, where a web service offers a REST API that uses several drivers, each of them connecting to the supported PaaS providers. The architecture is completed with the API client, which is used by the CloudSocket PaaS Abstraction Layer.

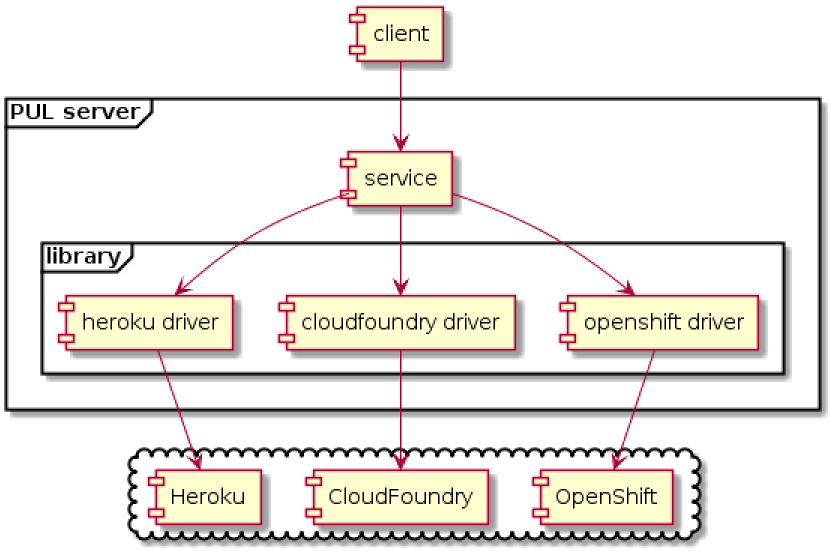


Figure 23 – Old architecture of the PaaS Unified Library.

Since each driver uses the respective Java client for a provider, it could happen, when more providers are added, that their dependencies collide. For this reason, the PUL has been redesigned to accept the following architecture, where there are several PUL instances, each of them managing a predefined list of providers (for example, one provider per instance), and a reverse proxy in front of them (for example, an NGINX). This new architecture is shown in Figure 24.

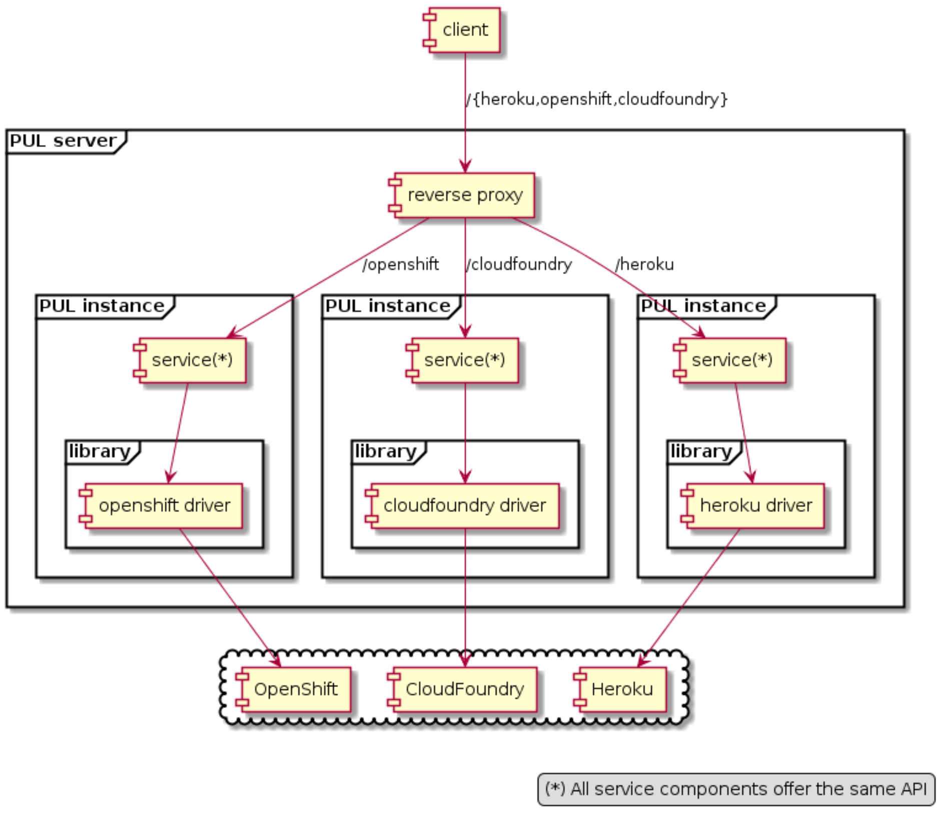


Figure 24 - Updated architecture of the PaaS Unified Library.

4.1.3.3 Setup

A technical setup description can be found in the CloudSocket Wiki[21].

4.1.3.4 Future Work

The immediate future work for the PaaS Unified Library is driven by the features and improvements requested by the PaaS Abstraction Layer. This is a list of possible enhancements:

- Transparent service binding. Currently, the name given to a service by the provider is needed to bind an application to a service (e.g., a MySQL service is called ClearDB in Heroku). This improvement will allow the creation of a service using a generic name.
- Scaling. This will allow the creation of an application passing scaling properties, either horizontal or vertical.
- OpenShift v3. OpenShift v3 is being implanted in OpenShift Online and it is the next interesting API to support in the PaaS Unified Library.

4.2 Adaptation Management

In order to support the Adaptation Plans defined in the CAMEL extension, the Cloudiator components were enriched to be able to evaluate more complex rules. A respective documentation can be found in the project wiki.[22] This component is used in the context of the later described Synergic Cross-Layer Adaptation Framework.

Cloudiator remains reasoning free, except for simple scalability and monitoring rules. Instead, we put a reasoning layer on top of Cloudiator (see Figure 25). This abstraction layer will be handled as an external component that uses Cloudiator (i.e. the REST-API of Colosseum) to keep the pre-defined life-cycle of components in Cloudiator independent of the user-defined imperative life cycles that will be handled by the Adaptation Management.

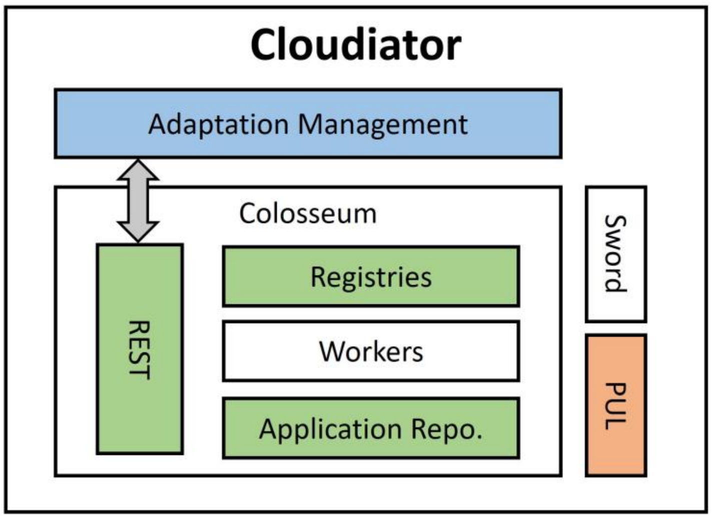


Figure 25 - The Cloudiator framework enriched by an adaptation management.

On the Cloudiator side, component adaptation focused on supporting the extended CAMEL definitions, i.e., we have created the super class *Action*, that is extended to cover the CAMEL capabilities added in the former described enhancements. The task composition described in CAMEL will not be handled by Cloudiator itself, but by the adaptation management layer. Cloudiator will implement in a first step the following atomic tasks: *Service Migration* and *Component Deployment*. Service Migration allows to specify a component as LifecycleComponent

[21] https://www.cloudsocket.eu/common-understanding-wiki/-/wiki/Main/Unified+PaaS+component
[22] https://www.cloudsocket.eu/common-understanding-wiki/-/wiki/Main/Adaptation+Management

and as PlatformComponent in order to be able to migrate from one provider to another. Component Deployment allows to add a component and its communication dependencies to an application. This action will lead to the change of the application's topology.

For this, the Adaptation Management provides a REST-API to allow the creation of the entities similar to the way defined in CAMEL, but with the concrete matches in the Cloudiator entities, i.e., the Adaptation Management component receives a Cloudiator-compatible description of CAMEL's *Adaptation Plans*.

4.3 Synergic Cross-Layer Monitoring Framework

In D3.3 [1], different BPaaS monitoring frameworks have been proposed. Two of them map to individual frameworks that have been developed by respective project partners, namely UULM and FORTH [11]. The last one maps to a synergic framework which attempts to combine the best of the worlds from the two aforementioned frameworks. To speed up the development time and facilitate the uptake of the respective monitoring framework in the CloudSocket implementation, it was decided to join forces and finally evolve the last of the frameworks, the synergic one. As such, the goal of this section is to provide a short summary of this framework, prescribe its evolution as well as provide guidelines on how this framework can be installed and run. In the end, some short term future work directions are provided which might be followed in the next forthcoming (6-month) period of the project.

4.3.1 Features

The synergic cross-layer monitoring framework exhibits a set of features that make it an ideal candidate for the realisation of the Monitoring Engine in the Execution Environment of the CloudSocket prototype. These features along with their benefits are the following:

- Coverage of three main layers, namely IaaS, SaaS and WfaaS. Monitoring mechanisms are provided for each individual layer. Moreover, layer-specific mechanisms subscribe to measurements on other lower layers in order to cover measurability gaps via the consideration of metric dependencies.
- Recovery capability in case a specific monitoring component becomes unavailable.
- Data redundancy through the capability to store the measurements sensed or derived in different measurement stores.
- Capability to cover the measurement of both domain-independent and domain-specific metrics. A set of domain-independent metrics is already supported while domain-specific metrics are developed on demand according to the respective use case that needs to be supported (and its respective domain of application). The capability to also support the measurement of cloud-specific and cross-cloud metrics is offered. In principle, this monitoring framework is able to measure any kind of metric, provided that its specification has been described in OWL-Q while the respective sensors which produce raw measurements needed for the computation of such a metric have been developed and can be incorporated in this framework.
- Capability to generate events which can be consumed by the *Adaptation Engine / Synergic Cross-Layer Adaptation Framework* (see Section 4.4) in order to trigger adaptation rules.
- Delivery of a publish-subscribe mechanism via which respective interested components can subscribe only to the corresponding metrics of interest. This can enable for example partitioning the adaptation functionality into different parts (i.e., instances of the Adaptation Engine) which can focus on the monitoring of a different set of metrics for load-balancing reasons.

4.3.2 Architecture

The architecture of the synergic monitoring framework is depicted in Figure 26. As it can be seen, it comprises the combination of the two individual monitoring frameworks of the two partners in a complementary way. In particular, the monitoring framework of UULM focuses on the monitoring at the IaaS level while the monitoring framework of FORTH focuses on the monitoring at the SaaS and WfaaS level. In case that specific IaaS measurements are needed by FORTH's framework, then that framework will subscribe to the corresponding metric on UULM's framework. In this sense, a publish-subscribe mechanism is employed to foster the communication between the two individual frameworks. Once a measurement is produced by either individual framework, it is published via the corresponding pub-sub mechanism to the *Evaluator* component which takes care of evaluating this measurement and mapping it to a respective event. The measurement evaluation is performed by exploiting the facilities of a CEP Engine. Once events are generated, they are published again via a pub-sub mechanism to the respective BPaaS adaptation component (*Synergic Cross-Layer Adaptation Framework*). The framework is accompanied via two components at the interface level: (a) a UI component which enables the inspection of measurements as well as the production of nice monitoring graphs; (b) a REST-based component which enables retrieving the measurements for a particular metric or metric context. Both of these components are actually part of UULM's Colosseum framework. The first maps to the Colosseum's UI and the second to Colosseum's REST API.

As described in Section 4.1.1, we introduced with the PlatformMonitor a new way to capture metrics from externally managed entities into Cloudiator that will then be available to other components. These advances the usability of the monitoring system and allows to create greater synergies between the monitoring and adaptation approaches of the different partners of the projects. We are currently implementing the most needed sensors to gather monitoring information from the most common PaaS providers and PlatformRuntimes.

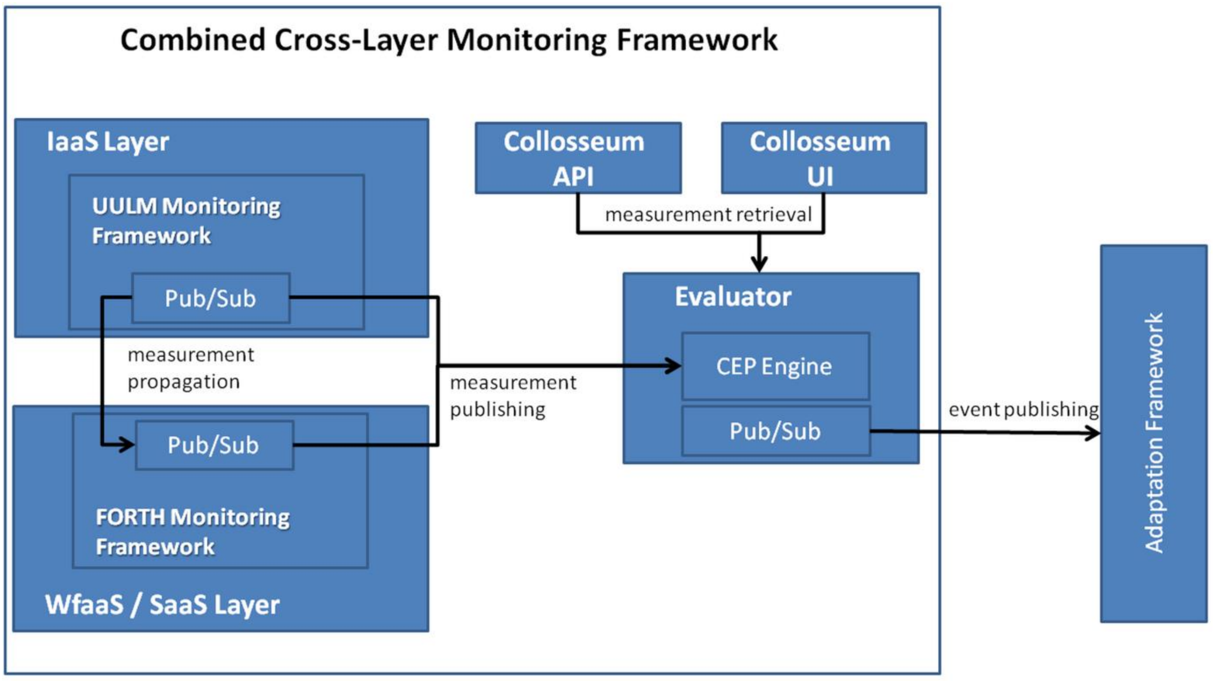


Figure 26 - The architecture of the synergic cross-layer monitoring framework

4.3.3 Setup

Each individual framework as well as the combined one map to Java projects which are situated in the UULM git repository[23]. Each project is maven-based such that respective maven commands can be used to compile and

[23] https://omi-gitlab.e-technik.uni-ulm.de/cloudsocket/cross_layer_monitoring

run it. The FORTH's monitoring framework is provided as a service such that it needs to be deployed in a respective servlet container like Tomcat. UULM's monitoring framework as well as the *Evaluator* component are incorporated in the overall Colosseum framework in order to also facilitate the respective integration with REST and visualisation functionalities. FORTH's monitoring framework needs to be configured in order to communicate properly with the UULM's monitoring framework. Both frameworks need also to be configured in order to communicate with the *Evaluator* component. The latter component should be also made aware of the presence of the aforementioned frameworks. All such information is actually configured via the use of respective configuration files whose content is first set before being actually read by the framework/components upon their initialisation. A more detailed manual concerning both the installation and usage of the synergic cross-layer framework can be found in the project wiki[24].

4.3.4 Future work

4.3.4.1 Dynamic monitoring

Based on the current capabilities that have been developed in the context of the PaaSage project, the UULM monitoring framework has the capability to update the monitoring infrastructure in case that there is a respective modification in the application deployment model. As this framework focuses on the IaaS layer and driven also by the need to support other types of adaptation at higher levels of abstraction, the FORTH's monitoring framework should be modified with the capability to modify its own part of the monitoring infrastructure, mostly related to modifications on the SaaS and WfaaS level. Apart from this, as also indicated in D3.3, it might also be interesting to realise a dynamic reconfiguration approach of the whole monitoring infrastructure which can also infer the optimal measurement periods for the metrics measured as well as be able to scale as needed. This is something we plan to investigate in the near future and certainly represents a very interesting research direction to follow.

4.3.4.2 Synergic Cross-Layer Monitoring Framework Evaluation

The proposed synergic framework needs to be validated based on the project use cases and evaluated according to various aspects, including monitoring performance and accuracy. Such validation and evaluation feedback can then unveil those optimisation points over the proposed framework that need to be followed.

4.3.4.3 PaaS Monitoring

The capability to deploy applications by also exploiting PaaS services is currently realised as an interesting extension over CAMEL and the Colosseum framework. However, in case such services are do exploited, there is a need to also monitor them or the respective facilities that they provide. In this sense, we need to explore all possible possibilities that such monitoring can take place and then select the best possible ones to be implemented as an extension to the current synergic monitoring framework developed. PaaS could for instance be exploited to obtain measurements for some certain metrics over respective BPaaS components or they could be used to deploy internal monitoring mechanisms over BPaaS components in order to enable the capability to properly monitor them or the underlying infrastructure.

4.4 Synergic Cross-Layer Adaptation Framework

Similarly, to the case of the cross-layer monitoring framework, the synergic cross-layer adaptation blueprint has been adopted to join forces, speed up the development process and propose a common framework which can also lead to attempt to achieve common publications. This framework's main features are now summarised, followed by a detailed analysis over its architecture and the respective procedures to be followed for its installation. Some short term future work directions are supplied in the end.

[24] https://www.cloudsocket.eu/common-understanding-wiki/-/wiki/Main/Synergic+Cross-Layer+Monitoring+Framework

4.4.1 Features

The synergic cross-layer adaptation framework attempts to combine the best of the worlds from the corresponding individual frameworks from UULM and FORTH [12]. The end result is a combined framework which exhibits the following list of features:

- Adaptation capabilities realised in different layers. Currently, scaling is supported at the IaaS level and service replacement at the SaaS level but other adaptation capabilities are planned to be developed in the near future also spanning the workflow level.
- Capability to specify complex adaptation rules in order to cover advanced adaptation scenarios.
- Capability to orchestrate individual adaptation capabilities at different levels to jointly confront a certain problematic situation.
- Combined with respective capability to be delivered by the Evaluation Environment, the adaptation rules monitored and triggered can be updated and enriched by possibly employing a learning approach.

All of the above features cater for the addressing of advanced adaptation scenarios which cannot be addressed via performing individual adaptation actions. As such, this feature agglomeration gives rise to a framework that advances the state-of-the-art in (cloud) service adaptation. Such a framework is ideal to realise the *Adaptation Engine* functionality in the respective CloudSocket prototype implementation. In addition, it can be considered a standalone component which could be adopted and provide added-value to any kind of (cloud) service management framework.

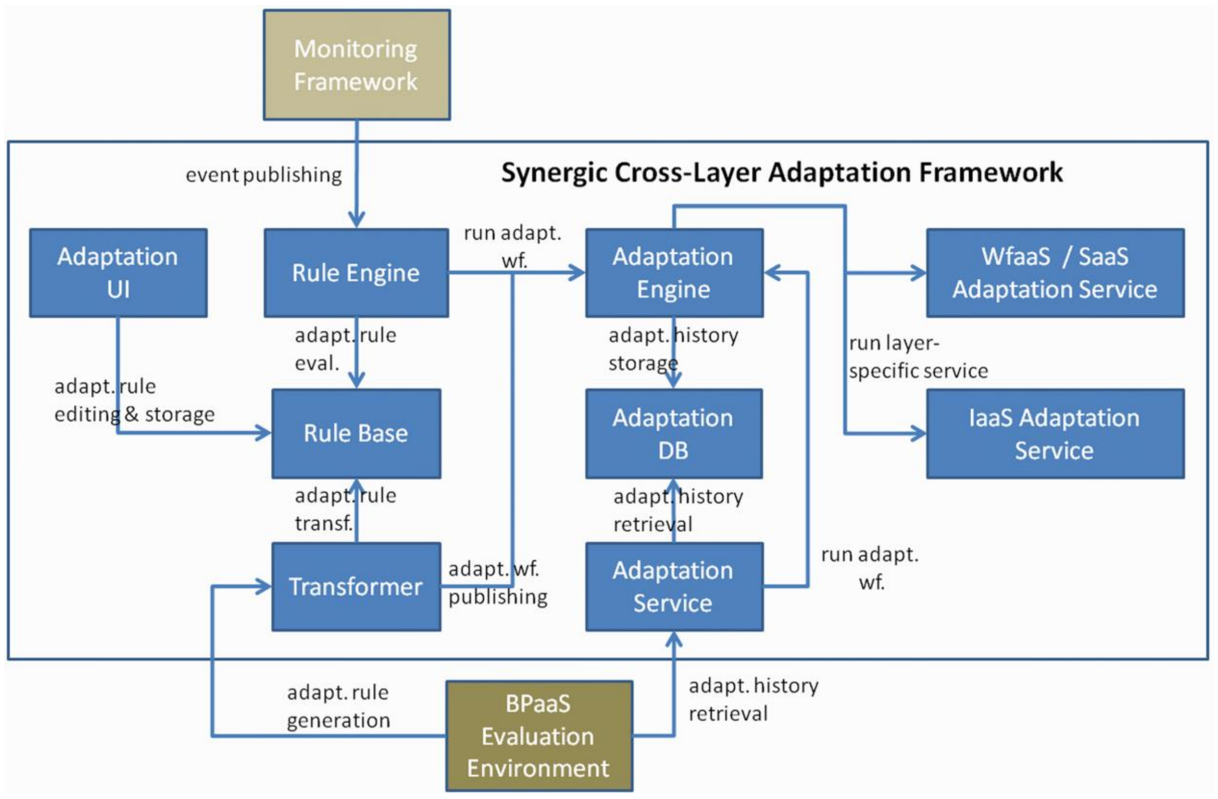


Figure 27 - The architecture of the synergic cross-layer adaptation framework

4.4.2 Architecture

The architecture of this component is simplified with respect to the one specified in Deliverable D3.3. This simplification has actually been performed by moving the adaptation rule generation functionality to the BPaaS Evaluation Environment (addressed in T3.3 and respective deliverables D3.5 [7] and D3.6). The respective simplified synergic cross-layer adaptation framework is depicted in Figure 27. As it can be seen, the framework comprises 9 main components:

- The *Rule Engine* is responsible for retrieving events and assessing whether a respective adaptation rule needs to be triggered.
- A triggered adaptation rule leads to invoking the *Adaptation Engine* which is a normal workflow engine with all adaptation strategies of rules deployed in the form of adaptation workflows. As such, the invocation of the Adaptation Engine just leads to creating an instance of an adaptation workflow and executing it.
- The *Transformer* is responsible for transforming the adaptation strategies of respective rules being suggested by the BPaaS Evaluation Environment into adaptation workflows. This maps to the ability to process the *AdaptationTask* part of CAMEL (by assuming that adaptation rules are specified by this language) and to transform it into a workflow in the BPMN format. The same component is also used to transform the adaptation rule into a rule which is compatible with the current Rule Engine implementation. The transformed rule is stored into the *Rule Base* of the *Rule Engine*.
- An adaptation workflow comprises executing adaptation services which are offered at different levels of abstraction. Two services are currently offered: one being able to support the adaptation of IaaS services (*IaaS Adaptation Service*) and another able to support the adaptation at the SaaS and workflow level (SaaS/WfaaS Adaptation Service). The first service maps to the Colosseum framework as this framework actually includes the capability to manage the lifecycle of application component as well as the respective IaaS offerings providing support to this lifecycle. Currently, this framework includes the realisation of scaling capabilities but also migration and bursting ones are under way, as indicated in Section 4.2. The second service is realised in the form of an add-on over a *Workflow Engine,* which is currently able to realise the service replacement adaptation action. This realisation is enabled by exploiting first the facilities of the *Smart Service Discovery and Composition Tool* in order to find the service replacement so as to produce a new BPaaS workflow for the running instance and then those of the Workflow Engine in order to migrate this instance to this new workflow.
- Once an adaptation workflow is successfully or not executed, the *Adaptation DB* is updated with the corresponding entry which stores specific information about this execution, such as how long it took to be executed, whether it has been successful, which adaptation rule was triggered and which BPaaS components have been affected. All this information can then be collected by the Evaluation Environment in order to support the functionality of adaptation rule derivation by employing a learning based approach.
- The *Adaptation UI* enables the visualisation and the management of the adaptation rules to be stored and executed in this synergic adaptation framework. Such a UI could exploit the concrete syntax of CAMEL in order to enable the visualisation and editing of the rules. More importantly, via the editing of rules, the human expert enters the loop and can be used to increase the suitability and appropriateness of the rules that are automatically generated by the system while providing in the beginning the initial input in the form of simple adaptation rules (in the form of simple event mapping to one adaptation action).
- The *Adaptation Service* enables the retrieval of adaptation-related information to suit the purposes of the Evaluation Environment while also catering for the programmatic triggering of adaptation rules in case there is a need to do so. Adaptation-related information includes a list of BPaaS-specific or generic cross-BPaaS rules and the adaptation history for a certain BPaaS or a certain adaptation rule. All this information is of course drawn from the corresponding *Adaptation DB*.

4.4.3 Setup

The adaptation framework code is split into various Java projects in UULM's git and the github repositories.[25,26,27] The *IaaS Adaptation Service* maps to the Colosseum framework so there is no need to further elaborate on it. The UI component has not been implemented yet. All relevant projects are maven-based such that they can be compiled and run via maven. The service-based components need to be associated to a war file which has to be deployed on a respective servlet container, like tomcat. A more detailed manual concerning this component is available in the project wiki[28].

4.4.4 Future work

4.4.4.1 Adaptation Action Coverage Extension

Currently, the adaptation capabilities of the synergic framework are limited to horizontal scaling at the IaaS level and service replacement at the SaaS level. Such capabilities can be considered as quite basic, inherent in other related adaptation frameworks, that can support a basic but sufficient number of simple scenarios. However, we believe that these capabilities need to be enhanced in order to enable the support of more advanced adaptation scenarios which can include either new adaptation capabilities or the orchestrated execution of existing ones. To this end, we plan to support in the near future new adaptation actions spanning: (a) cloud bursting: capability to reserve resources from a public cloud when there are no more resources left in the private cloud initially exploited; (b) vertical scaling: in some cases, vertical scaling might be less expensive than horizontal one but still be able to confront the current problematic situation; (c) workflow modification: a BPaaS workflow could be modified either by the broker or the BPaaS client (in cases it possess the respective expertise). There are two situations where the latter can occur: (i) emergency cases where some workflow tasks might need to be skipped or re-ordered; (ii) cases where a human worker decides to split one user task into two as well as delegate the execution of the original task or the new sub-tasks to other workers; (d) cloud migration: we are currently exploring the migration of special types of components, like DBs. By including all these new capabilities in the current support list, the adaptation portfolio is enlarged while the respective possibilities to confront a current problematic situation are actually multiplied, thus enabling our framework to deliver quite sophisticated and advanced cross-layer adaptation functionality.

4.4.4.2 PaaS Support

Driven also by the respective extension of the deployment capabilities of the framework to cover the exploitation of PaaS services, we foresee that in the near future a PaaS service could be utilised in the following two ways: (a) as a potential substitute or alternative to an IaaS service. In this sense, a PaaS could be exploited in migration or bursting scenarios; (b) as a potential service provider for adaptation capabilities which are of course restrained in a certain cloud. As such, a PaaS could be regarded as an adaptation service which could for instance trigger the scaling of a BPaaS component or its migration to a new VM. To this end, we will explore these two possibilities and possibly extend the current adaptation functionality to include at least one of them, if not both.

4.4.4.3 Synergic Adaptation Framework Validation & Evaluation

Similarly, to the case of the synergic cross layer monitoring framework, we plan to validate the adaptation framework based on the CloudSocket use cases as well as evaluate it according to particular evaluation aspects, such as the adaptation performance and accuracy/suitability. The validation and evaluation feedback will pave the

[25] https://github.com/cloudiator/axe-aggregator
[26] https://omi-gitlab.e-technik.uni-ulm.de/cloudsocket/adaptation-management
[27] https://omi-gitlab.e-technik.uni-ulm.de/cloudsocket/cross_layer_adaptation
[28] https://www.cloudsocket.eu/common-understanding-wiki/-/wiki/Main/Synergic+Cross-Layer+Adaptation+Framework

way for further improving the synergic adaptation framework either through the correction of some implementation issues or the update over some adaptation capabilities.

5 CONCLUSION AND FUTURE WORK

This chapter gives a short summary of the deliverable and an outlook to the further prototype development.

5.1 Summary

This report describes the research prototypes for BPaaS cloud modelling, allocation and execution. These prototypes will be part of the BPaaS Allocation Environment and BPaaS Execution Environment.

The prototypes implement

(a) advanced cloud service modelling approaches with CAMEL and OWL-Q,
(b) a well-integrated Smart Service Discovery and Composition prototype,
(c) a supportive modelling approach combining CAMEL and DMN,
(d) an extended cloud orchestration platform based on the Cloudiator tool-suite and
(e) a synergic cross-layer monitoring and adaptation framework.

Some of the presented prototypes will later be used in the prototypes described in D3.5 [7] BPaaS Monitoring and Evaluation Blueprints, such as the Monitoring Framework that provides input for the later evaluation. Furthermore, some prototypes will be integrated into the stable environment developed in WP4, whereas the others will remain in the research environment that is constantly updated with the latest features in terms of the Continuous Integration process described in T4.5.

The prototypes are available free for download from the CloudSocket webpage[29].

5.2 Future Work

These prototypes must be further evaluated in terms of their suitability for other real-life use cases. This will be performed via more detailed studies. Once this is fulfilled, we will bring selected prototypes in a more mature state.

Cloudiator for example aims to be a production-ready solution and therefore demands for highly stable modules. This implies that continuous quality assurance will be applied by the future developers of Cloudiator, once the prototypes are integrated into the stable master branch.

The presented prototype towards DMN-to-CAMEL mapping will be further elaborated with respect to being integrated into the currently running BPaaS Allocation Environment in order to ease the allocation process. Also it will be better integrated in modelling tools such as ADOxx. The Smart Service Discovery and Composition prototype could be also integrated in the BPaaS Allocation Environment to enable taking allocation decisions based on technical requirements.

The cross-layer monitoring and adaptation frameworks seem to be good replacements for the Monitoring and Adaptation Engines in the stable CloudSocket environment. In this sense, they could be integrated with the latter environment, once their development has been finalised.

Based on the above, many of the prototypes are currently in an integration process with WP4 and will further serve the implementation and demonstration purposes of the CloudSocket project.

[29] https://www.cloudsocket.eu/download

6 REFERENCES

[1] D. Seybold, K. Kritikos, F. Griesinger, Hinkelmann, K. Kritikos, R. Sosa, J. Iranzo and C. Zeginis, ‘D3.3 – BPaaS Allocation and Execution Environment Blueprints’, CloudSocket project deliverable, June 2016

[2] J. Jähnert et. al., ‘D5.1 – Initial CloudSocket Setup Report’, CloudSocket project deliverable, December 2015

[3] J. Iranzo et al., ‘First BPaaS Prototype - D4.2 - D4.3 - D4.4’, CloudSocket project deliverable, July 2016.

[4] Kyriakos Kritikos, Dimitris Plexousakis: Semantic SLAs for Services with Q-SLA. CF 2016.

[5] Kyriakos Kritikos, Dimitris Plexousakis: Towards Combined Functional and Non-functional Semantic Service Discovery. ESOCC 2016: 102-117.

[6] Kyriakos Kritikos, Kostas Magoutis, Dimitris Plexousakis: Towards Knowledge-Based Assisted IaaS Selection. CloudCom 2016.

[7] K. Kritikos, 'D3.5 - BPaaS Monitoring and Evaluation Blueprints'. CloudSocket project deliverable, December 2016.

[8] Kyriakos Kritikos, Dimitris Plexousakis: Multi-cloud Application Design through Cloud Service Composition. CLOUD 2015: 686-69.

[9] Lam, J. S. C., Vasconcelos, W. W., Guerin, F., Corsar, D., Chorley, A., Norman, T. J., ... Nieuwenhuis, K. (2009). ALIVE: a framework for flexible and adaptive service coordination. In H. Aldewereld, V. Dignum, & G. Picard (Eds.), Engineering Societies in the Agents World X: 10th International Workshop, ESAW 2009 Utrecht, The Netherlands, November 18-20, 2009. Proceedings. (pp. 236-239). (Lecture notes in computer science; Vol. 5881). Berlin: Springer.

[10] G. Baryannis and D. Plexousakis, ‘Fluent calculus-based semantic web service composition and verification using wssl’, in Service-Oriented Computing–ICSOC 2013 Workshops, 2013, pp. 256–270.

[11] C. Zeginis, K. Kritikos, P. Garefalakis, K. Konsolaki, K. Magoutis, and D. Plexousakis, ‘Towards cross-layer monitoring of multi-cloud service-based applications’, in Service-Oriented and Cloud Computing, Springer, 2013, pp. 188–195.

[12] Zeginis, C., Kritikos, K., Plexousakis, D. (2014). Event Pattern Discovery for Cross-Layer Adaptation of Multi-cloud Applications. In ESOCC, 138-147.

[13] Group, O.M.: Decision model and notation. Tech. rep., OMG, http://www.omg.org/spec/DMN/1.1/ (2016)

[14] Griesinger, Frank et al.: A DMN-based Approach for Dynamic Deployment Modelling of Cloud Applications; Service Oriented and Cloud Computing, ESOCC 2016, September 2016

ANNEX A: CAMEL FRAGMENT

Deployment model fragment for the ChristmasCardDesginer, cloud provider = UULM, customer level = gold

```
deployment model DeploymentPlan {

                requirement set DemoAppRequirement {
                        os: Requirement.UbuntuOS
                }

                vm DemoAppVM {
                        requirement set DemoAppRequirement
                        provided host DemoAppCompononentHost
                }


                internal component DemoAppComponent {
                        provided communication WebServiceCommunication {
port: 2181 }
                        required host DemoAppHostReq

                        configuration DemoAppConfiguration {
                                download: 'sudo apt-get install -y curl'
                                install: 'curl -o
demoapp_checkstart_ubuntu.sh https://omi-gitlab.e-technik.uni-
ulm.de/cloudsocket/prototype_v1_files/raw/master/demoapp_checkstart_ubuntu.
sh && chmod +x demoapp_checkstart_ubuntu.sh && curl -o
demoapp_install_ubuntu.sh https://omi-gitlab.e-technik.uni-
ulm.de/cloudsocket/prototype_v1_files/raw/master/demoapp_install_ubuntu.sh
&& chmod +x demoapp_install_ubuntu.sh && ./demoapp_install_ubuntu.sh'
                                configure: ''
                                start: '/apache-tomcat-7.0.65/bin/catalina.sh
run'
                 upload: 'source /demoapp_checkstart_ubuntu.sh'
                        }
                }


                hosting DemoAppToDemoAppVM {
                        from DemoAppComponent.DemoAppHostReq to
DemoAppVM.DemoAppCompononentHost
                }


                internal component instance DemoAppComponentInstance typed
BundleCamelModel.DeploymentPlan.DemoAppComponent{
                        required host instance DemoAppInstanceHostReq typed
DemoAppComponent.DemoAppHostReq
                        provided communication instance
DemoAppWebServiceCommunication typed
DemoAppComponent.WebServiceCommunication
                }

                ###### large VM selected for GOLD customer
                vm instance DemoAppOmistackSmallInstance typed
BundleCamelModel.DeploymentPlan.DemoAppVM {
```

```
                        vm type:
BundleCamelModel.OmistackProvider.Omistack.VM.VMType
                        vm type value:
BundleCamelModel.OmistackType.VMTypeEnumeration.m1.small
                        provided host instance DemoAppComponentHostInstance
typed DemoAppVM.DemoAppCompononentHost
                }

                # defines the host instance model from component instance to
vm instance
                host DemoAppComponentInstance.DemoAppInstanceHostReq on
DemoAppOmistackSmallInstance.DemoAppComponentHostInstance typed
BundleCamelModel.DeploymentPlan.DemoAppToDemoAppVM


}
```

ANNEX B: LIST OF TOOLS

ADONIS®	Business Process modelling tool, http://www.boc-eu.com/,
CAMEL	A cloud domain specific language, http://camel-dsl.org/,
Cloudiator	Cloud orchestration tool-suite, https://cloudiator.github.io/,
DMN	Decision Model and Notation, OMG standard, http://www.omg.org/spec/DMN/,
jclouds	IaaS cloud abstraction library, https://jclouds.apache.org/,
nginx	Web server, http://nginx.org/,
Protégé	ontology editor, http://protege.stanford.edu/,
Tomcat	Java-based application server, http://tomcat.apache.org/,
Visor	Monitoring agent of the Cloudiator framework, https://github.com/cloudiator/visor/,

ANNEX C: CAMEL MODELS

In this appendix, we provide the complete CAMEL models in XMI form for the PaaS deployment (Section 2.1.1), cloud bursting (Section 2.3.2.1) and service replacement scenarios of the Christmas Card Sending use case (Section 2.3.2.2). The examples can be found in the wiki[30].

PaaS Deployment CAMEL Model

The example for the PaaS deployment can be found in the wiki[31]. An excerpt of the internal component with the new PaaS configuration is shown here:

```
<deploymentModels
    name="ChristmasCardSendingDeployment">
  <internalComponents
      xsi:type="deployment:InternalServiceComponent"
      name="CardDesigner"
      type="SERVLET"
      serviceType="REST">
    <configurations
        xsi:type="deployment:PaaSConfiguration"
        name="CardDesignerPaaSConfiguration"
        api="PUL"
        version="1.0"
        endpoint=""/>
    <requiredHost
        name="CardDesignerRequiredHost"/>
    <workflowTaskIDs>sid-1C5E88D8-C7E3-4FAC-85E5-FAD80099C28B</workflowTaskIDs>
    <workflowTaskIDs>sid-D891CCD3-40B6-4555-9C84-81BECCDCB1C2</workflowTaskIDs>
    <workflowTaskIDs>sid-B99151C2-9914-4E71-9540-9DFE38334B7B</workflowTaskIDs>
  </internalComponents>
```

Cloud Bursting CAMEL Model

The example for the cloud bursting can be found in the wiki[32]. An excerpt of such an adaptation rule can be seen here:

```
<adaptationModels
    name="ChristmasCardSendingAdaptation">
  <rules name="BURST"
      event="//@adaptationModels.0/@events.2"
      task="//@adaptationModels.0/@tasks.0"
      entity="//@organisationModels.0/@organisation"/>
  <events
      xsi:type="adaptation:NonFunctionalEvent"
      name="ViolRTCondition"
      metricCondition="//@metricModels.0/@conditions.0"
      isViolation="true"/>
  <events
```

[30] https://www.cloudsocket.eu/common-understanding-wiki/-/wiki/Main/Examples
[31] https://www.cloudsocket.eu/common-understanding-wiki/-/wiki/Main/PaaS+Deployment
[32] https://www.cloudsocket.eu/common-understanding-wiki/-/wiki/Main/Cloud+Bursting

```
            xsi:type="adaptation:NonFunctionalEvent"
            name="ViolCapacityCondition"
            metricCondition="//@metricModels.0/@conditions.1"
            isViolation="true"/>
        <events
            xsi:type="adaptation:BinaryEventPattern"
            name="BurstEventPattern"
            leftEvent="//@adaptationModels.0/@events.0"
            rightEvent="//@adaptationModels.0/@events.1"/>
        <tasks xsi:type="adaptation:ComponentDeployment"
            name="EmailServiceOnPublicCloud"
            components="//@deploymentModels.0/@internalComponents.1"
            vm="//@deploymentModels.0/@vms.2"/>
    </adaptationModels>
```

Service Replacement CAMEL Model

The example for the service replacement can be found in the wiki[33]. An excerpt of this adaptation rule is shown here:

```
    <adaptationModels
        name="ChristmasCardSendingAdaptation">
        <rules name="Replace"
            event="//@adaptationModels.0/@events.0"
            task="//@adaptationModels.0/@tasks.0"
            entity="//@organisationModels.0/@organisation"/>
        <events
            xsi:type="adaptation:NonFunctionalEvent"
            name="ViolRTCondition"
            metricCondition="//@metricModels.0/@conditions.0"
            isViolation="true"/>
        <tasks xsi:type="adaptation:ServiceReplacement"
            name="ReplaceEmailService"
            previousService="//@deploymentModels.0/@internalComponents.1"/>
    </adaptationModels>
```

[33] https://www.cloudsocket.eu/common-understanding-wiki/-/wiki/Main/Service+Replacement