

BPAAS ALLOCATION AND EXECUTION ENVIRONMENT BLUEPRINTS

D3.3

Editor Name	Daniel Seybold (UULM)
Submission Date	June 30, 2016
Version	1.0
State	FINAL
Confidentially Level	PU



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

EXECUTIVE SUMMARY

The first phase of the BPaaS lifecycle - the BPaaS Design - and the respective research challenges were covered in D3.1 "Modelling Framework for BPaaS". The first phase produces the BPaaS Design Package that provides the input for the following phases. This document introduces research challenges on the second and third phase of the BPaaS lifecycle, which are supported by the BPaaS Allocation and Execution Environments.

This is the second deliverable of work package 3 "BPaaS Allocation and Execution Environment Blueprints". Its content is twofold: first, in the Allocation phase, the mapping of the abstract workflows from the Design Package to executable workflows that involves incorporating actual cloud services to realise and support the workflow functionality. The executable workflow along with additional information as SLAs and scalability rules then constitute the BPaaS Bundle produced. Second, the execution of the BPaaS Bundle, including the orchestration, monitoring and adaptation of all services involved.

In order to enhance the BPaaS lifecycle with research findings, three Blueprint categories are covered within this deliverable: The BPaaS Modelling Blueprints, the BPaaS Allocation Blueprints and the BPaaS Execution Blueprints. Each Blueprint comprises a set of research assets. The upcoming deliverable D3.4 "BPaaS Allocation and Execution Environment Prototypes", which is due in December 2016, will build upon these research assets and provide prototypes for each Blueprint category.

The BPaaS Blueprint comprises the modelling approach for the BPaaS Bundle. In order to provide complete support to the BPaaS allocation and execution phases, the CAMEL domain-specific language is exploited and extended as one research asset in order to additionally cover the modelling of SaaS and PaaS allocation decisions apart from IaaS ones. Apart from the modelling of the application deployment, CAMEL also covers the aspects of application adaptation and monitoring. The second research asset is the semantic quality description language OWL-Q that can be used to describe non-functional requirements and capabilities at all levels of abstraction as well as semantic hierarchical SLAs. The resulting assets are (1) *PaaS/SaaS support of CAMEL* and (2) *SLA support in OWL-Q*

The BPaaS Allocation Blueprint presents more sophisticated service selection and concretisation algorithms to map the abstract workflows to actual cloud services. This comprises smart service discovery, composition and selection tools. Based on the semantically lifted BPaaS Design Package and the Allocation Environment Blueprint, a more accurate creation of the corresponding part of the BPaaS Bundle, i.e., the executable and deployable workflow is enabled. The resulting assets are (3) *Smart Service Discovery and Composition Tools* and (4) *DMN to CAMEL Mapping*.

The BPaaS Execution Blueprint is split into three crucial sub-phases of BPaaS execution, i.e., orchestration, monitoring and adaptation with the offering of corresponding research assets for each. BPaaS orchestration encompass research assets regarding the BPaaS execution across different cloud service levels. The BPaaS monitoring assets focus on a self-scalable monitoring infrastructure (UULM) as well as on cross-layer monitoring (FORTH) framework while a synergic framework between these two is also proposed. The adaptation assets cover the scalability on the lower cloud service levels (UULM) as well as service adaptation in higher cloud service levels (FORTH) along with a corresponding proposal for a synergic/combined approach. The resulting assets are (5) *PaaS orchestration*, (6) *Dynamic IaaS Selection at Runtime*, (7) *Distributed and self-scalable Monitoring*, (8) *Cross-Layer Monitoring*, (9) *Synergic Cross-Layer Monitoring*, (10) *AXE Adaptation Framework*, (11) *Cross-Layer Adaptation* and (12) *Synergic Cross-Layer Adaptation*

All three Blueprints and the included research assets are categorised according to their added value for the BPaaS lifecycle and their current state. This facilitates the prioritisation for work package 4 to derive the desired features from the architectural perspective. Based on the prioritisation, the prototype development is structured for the follow up deliverable D3.4.

PROJECT CONTEXT

Workpackage	WP3: Business Process as a Service Research
Task	T3.2: BPaaS Allocation and Execution Environment Research
Dependencies	Input to D3.4, T3.3 and WP4

Contributors and Reviewers

Contributors	Reviewers
Daniel Sebold, Frank Griesinger, Jörg Domaschka (UULM) Kyriakos Kritikos, Chrysostomos Zeginis (FORTH) Román Sosa, Joaquin Iranzo Yuste (ATOS)	Radu Davidescu (YMENS) Simone Cacciatore (FHOSTER) Román Sosa (ATOS)

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

Version History

Version	Date	Authors	Sections Affected
0.1	May 02, 2016	Daniel Seybold (UULM)	Initial version, TOC
0.2	May 04, 2016	Daniel Seybold, Kyriakos Kritikos (FORTH)	Updated TOC
0.3	May 20, 2016	Daniel Seybold (UULM), Kyriakos Kritikos (FORTH), Chrysostomos Zeginis (FORTH), Román Sosa (ATOS), Joaquin Iranzo Yuste (ATOS)	All sections
0.4	May 31, 2016	Daniel Seybold (UULM)	Finalised Monitoring section
0.5	June 3, 2016	Daniel Seybold (UULM)	Introduction, Summary
0.6	June 15, 2016	Daniel Seybold (UULM), Frank Griesinger (UULM), Kyriakos Kritikos (FORTH)	Blueprint revisions, all sections
0.7	June 17, 2016	Daniel Seybold (UULM), Frank Griesinger (UULM), Kyriakos	Consolidated internal review version




		Kritikos (FORTH), Chrysostomos Zeginis (FORTH)	
0.8	June 27, 2016	Daniel Seybold (UULM), Simone Cacciatore (FHOSTER) Roman Sosa (AT OS) Alexandru Ganga (YMENS)	Integrated internal review feedback
0.9	June 29, 2016	Daniel Seybold (UULM) , Kyriakos Kritikos (FORTH)	Final improvements
1.0	June 30, 2016	Daniel Seybold (UULM)	Finalised document

Copyright Statement – Restricted Content

This document does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of its content.

This is a restricted deliverable that is provided to the community under the license Attribution-NonCommercial-ShareAlike 3.0 Unported defined by creative commons <http://creativecommons.org>

You are free:

	to share within the restricted community— to copy, distribute and transmit the work within the restricted community
Under the following conditions:	
	Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
	No Derivative Works — You may not alter, transform, or build upon this work.

With the understanding that:

Waiver — Any of the above conditions can be waived if you get permission from the copyright holder.

Other Rights — In no way are any of the following rights affected by the license:

- Your fair dealing or fair use rights;
- The author's moral rights;
- Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

Notice — For any reuse or distribution, you must make clear to others the license terms of this work.

This is a human-readable summary of the Legal Code available online at:

<http://creativecommons.org/licenses/by-nd/3.0/>

TABLE OF CONTENT

1	Introduction and Problem Statement	12
1.1	Project Context.....	13
1.2	Research Problem	13
1.3	Structure.....	15
2	BPaaS Modelling Blueprint.....	16
2.1	State-of-the-art.....	16
2.1.1	Service Description Languages	17
2.1.2	Cloud Service Modelling.....	18
2.2	CAMEL.....	19
2.2.1	Original Version	20
2.2.2	Extension: SaaS Modelling.....	22
2.2.3	Extension: Cross-Layer Description of Components.....	23
2.2.3.1	Configuration per service level and ability.....	23
2.2.3.2	Overloading the component configuration.....	24
2.2.3.3	Crushed configurations by building blocks.....	25
2.2.3.4	Conclusion.....	27
2.3	OWL-Q.....	28
2.3.1	Original Version	28
2.3.2	Extensions.....	30
2.3.2.1	Core Extensions.....	30
2.3.2.2	SLA Extension.....	33
2.4	Future Work.....	36
2.4.1	CAMEL Adaptation.....	36
2.4.2	CAMEL Semantic Annotations.....	37
3	Allocation Environment Blueprint.....	38
3.1	State-of-the-Art	39
3.2	Smart Service Discovery & Composition.....	40
3.2.1	Smart Service Discovery	40
3.2.1.1	Non-Functional Service Discovery.....	40
3.2.1.1.1	Prototype Architecture.....	41
3.2.1.1.2	Mixed Category Algorithms	42
3.2.1.1.3	Ontology-based Category of Algorithms.....	44
3.2.1.1.4	Overall Discussion on Non-Functional Matchmaking Algorithms	44
3.2.1.2	Functional Service Matchmaking.....	45

3.2.1.3	IaaS Matchmaking.....	46
3.2.2	Smart Functional Service Composition.....	46
3.3	Simultaneous IaaS & SaaS Service Selection Algorithm.....	46
3.4	DMN to CAMEL mapping.....	49
3.4.1	DMN Mapping Scenario.....	50
3.4.2	Identified Challenges.....	52
3.5	Future Research.....	52
3.5.1	Combined Service Discovery.....	52
3.5.2	Overall Service Concretisation Method.....	52
3.5.3	QoS Mapping Derivation.....	52
3.5.4	PaaS Consideration in Discovery & Selection.....	53
3.5.5	Rich Service Specification.....	53
3.5.6	Formalism Transformation.....	53
3.5.7	Service Filtering.....	53
3.5.8	Semantic annotations for DMN Mapping.....	53
4	Execution Environment Blueprint.....	55
4.1	Orchestration.....	55
4.1.1	State of the art.....	56
4.1.1.1	IaaS Abstraction Tools and Platforms.....	56
4.1.1.2	PaaS Abstraction Tools and Platforms.....	57
4.1.2	Cloud Provider Engine (Cloudiator).....	59
4.1.2.1	Original Version.....	59
4.1.2.2	Evaluation.....	60
4.1.2.3	Extension: PaaS orchestration and abstraction layer.....	65
4.1.3	Future Research.....	68
4.1.3.1	Dynamic IaaS Selection at Runtime.....	68
4.2	Monitoring.....	68
4.2.1	State of the art.....	69
4.2.1.1	Quality Models.....	69
4.2.1.2	Service Monitoring.....	71
4.2.1.3	Cloud Monitoring.....	71
4.2.2	Scalability/ Elasticity Evaluation of Distributed Databases.....	74
4.2.2.1	Evaluated NoSQL databases.....	74
4.2.2.1.1	Apache Cassandra.....	74
4.2.2.1.2	Couchbase.....	75
4.2.2.1.3	MongoDB.....	75

4.2.2.2	Benchmarking Tool	75
4.2.2.3	Methodology.....	76
4.2.2.4	Results.....	78
4.2.2.4.1	Scalability Results.....	78
4.2.2.4.2	Elasticity Results.....	79
4.2.2.4.3	Conclusion.....	81
4.2.2.5	Future Evaluation Scenarios.....	81
4.2.3	UULM Approach	81
4.2.3.1	Monitoring Agent: Visor	82
4.2.3.2	Aggregation Levels.....	83
4.2.3.3	Distributed Architecture	83
4.2.3.4	Generic TSDB Layer.....	84
4.2.4	FORTH Approach.....	85
4.2.4.1	Distributed Cross-Layer Monitoring Framework.....	85
4.2.4.2	Cross-Layer Quality Model.....	88
4.2.5	Integration / Synergy of Approaches.....	91
4.2.6	Future Research.....	92
4.2.6.1	Cross-Layer Quality Model Expansion.....	92
4.2.6.2	Quality Model Realisation.....	92
4.2.6.3	Synergic Cross-Layer Monitoring Approach.....	93
4.2.6.4	Monitoring Adaptation.....	93
4.3	Adaptation	93
4.3.1	State of the Art	94
4.3.1.1	Service Monitoring & Adaptation.....	94
4.3.1.1.1	Cross-Layer Approaches.....	94
4.3.1.2	Languages for Adaptation Plans.....	95
4.3.2	UULM Approach	96
4.3.2.1	AXE	96
4.3.2.2	Adaptation Plans.....	97
4.3.3	FORTH Approach.....	98
4.3.4	Integration / Synergy of Approaches.....	100
4.3.5	Future research.....	100
4.3.5.1	Dynamic Adaptation Workflow Concretisation.....	100
4.3.5.2	Optimised Derivation of Adaptation Strategies.....	101
4.3.5.3	Layer-Specific Adaptation Action Realisation.....	101
5	Interaction with other environments	103

5.1	Required Input.....	103
5.2	Exploitable Output.....	104
6	Summary: Research showroom	106
6.1	Research assets.....	106
6.2	Blueprint handover process.....	110
6.3	Summaray and Future Work.....	111
7	References.....	112
Annex A:	List of Abbreviations.....	117

LIST OF FIGURES

- Figure 1 - Initial High-level Architecture of CloudSocket.....12
- Figure 2 - Identified Research Challenges14
- Figure 3 - BPaaS Modelling Blueprint.....16
- Figure 4 - Advanced class structure of the configurations of a component in CAMEL.....24
- Figure 5 - ConfigurationsRequirementSet part of the class structure.....25
- Figure 6 - Class structure for Container and Containerizables concept.....25
- Figure 7 - Provider model for Containers and example instances.....26
- Figure 8 - Example for a application stack from building blocks.....26
- Figure 9 - Simplified application description.....27
- Figure 10 - Five main OWL-Q facets.....31
- Figure 11 - The OWL-Q specification facet (with concepts coloured in blue).....32
- Figure 12 - Q-SLA sub-facet (with concepts coloured in light blue).....35
- Figure 13 - Allocation Environment Blueprint.....38
- Figure 14 - The architecture of the non-functional service matchmaking prototype.....41
- Figure 15 - An example subsumption hierarchy43
- Figure 16 - Integration points for DMN into the BPaaS process.....50
- Figure 17 - DMN-to-CAMEL mapping51
- Figure 18 - Overall architecture of the BPaaS Execution Environment.....55
- Figure 19 - Cloudiator architecture59
- Figure 20 - Unified Life-cycle Handling in the Cloud Provider Engine.....65
- Figure 21 - Cloudiator with IaaS and PaaS abstraction layer.....66
- Figure 22 - YCSB Architecture76
- Figure 23 - Benchmarking setups77
- Figure 24 - Apache Cassandra Elasticity Benchmark80
- Figure 25 - Couchbase Elasticity Benchmark80
- Figure 26 - MongoDB Elasticity Benchmark.....81
- Figure 27 - Visor.....82
- Figure 28 - Distributed Monitoring Architecture84
- Figure 29 - The logical architecture of FORTH's monitoring framework.....87
- Figure 30 - Physical architecture of the FORTH's cross-layer monitoring framework.....89
- Figure 31 - Overview of the cross-layer quality model90
- Figure 32 - Combined cross-layer monitoring architecture.....91
- Figure 33 - Example of an adaptation plan with component scaling and short-term service substitution98
- Figure 34 - FORTH's Adaptation Framework.....99
- Figure 35 - Research blueprint handover process 110

LIST OF TABLES

- Table 1 - The evaluation of Q-SLA against more representative state-of-the-art SLA languages.....34
- Table 2 - Cloud Deployment Type DT51
- Table 3 - Cloud Orchestration Tool Comparison.....61
- Table 4 - Life-cycle Actions of the Generic PaaS Deployment APIs67
- Table 5 - YCSB workloads CRUD ration in %.....77
- Table 6 - create-only workload results78
- Table 7 - read-update workload results.....78
- Table 8 - read-heavy workload results79
- Table 9 - Aggregation Levels83
- Table 10 - TSDB feature comparison.....85
- Table 11 - BPaaSModelling Blueprint assets..... 106
- Table 12 - Allocation Environment Blueprint assets 107
- Table 13 - Execution Environment Blueprint assetss 110

1 INTRODUCTION AND PROBLEM STATEMENT

This document introduces research Blueprints with respect to the BPaaS Allocation and BPaaS Execution Environment (cf. Figure 1). The Blueprints focus on the BPaaS Allocation and Execution Environment research challenges and solutions in the context of CloudSocket. The previous Deliverable D3.1 has focused on the BPaaS Design Environment and its outcome constitutes the starting point of this document, respectively the input to the Allocation Environment. This deliverable is concerned with the mapping of business episodes to deployable solutions in the cloud, which are then taken care of accordingly by enabling their adaptive provisioning. As such, the Blueprints presented focus on the specification of the appropriate information in order to support the envisioned activities (deployment, execution, monitoring & adaptation) as well as on the realisation of such activities.

As a blueprint has a meaning of a plan and not an actual realisation, this deliverable actually explains the analysis of concrete algorithms or frameworks that support the aforementioned activities. In this sense, it goes far deeper than reporting some blueprints. As such, we adopt a different term to refer to these algorithms or frameworks, which is a research asset. This is also more close to our final goal, i.e., to introduce particular assets or components that could be adopted by the CloudSocket implementation by providing add-ons to existing components or more advanced replacements of them. Nevertheless, in some cases, some algorithms/frameworks are just a sketch of an idea. Moreover, additional algorithms are sketched in future work directions. In this case, indeed, such ideas or sketches can be regarded as blueprints that can be realised in the next 6 months in the project such that they can then constitute more mature research assets that could be exploited by the CloudSocket implementation.

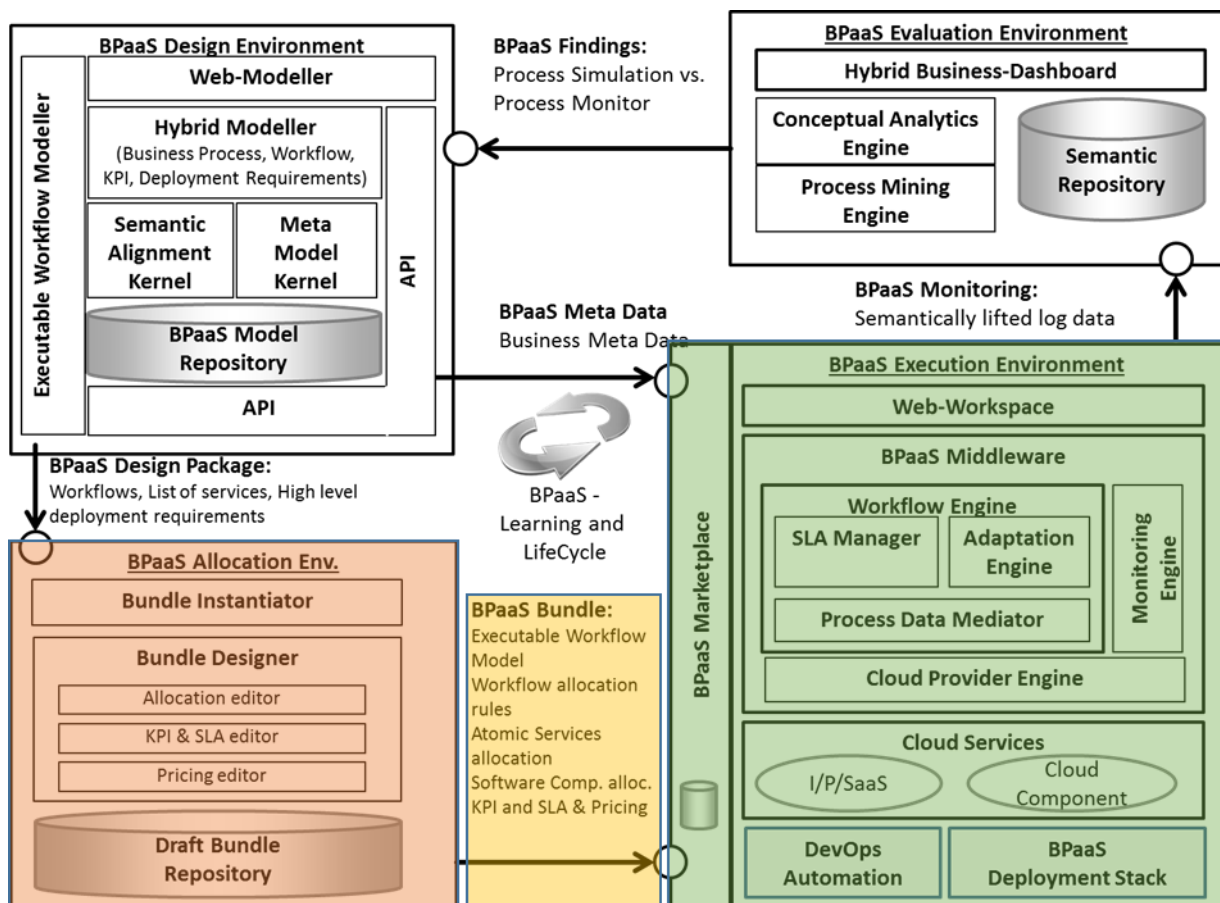


Figure 1 - Initial High-level Architecture of CloudSocket

1.1 Project Context

As this document focuses on the Allocation and Execution Environment, a high-level architecture of all environments, which constitute the CloudSocket prototype, with their main components is shown in Figure 1. A detailed architectural overview of all CloudSocket environments can be found in Deliverable D4.1 [1]. The BPaaS Allocation Environment allows a CloudSocket Broker to retrieve workflows from the BPaaS Design Environment and create a Cloud deployable as well as executable Workflow Bundle – named as BPaaS Bundle - and publish it in the Marketplace, by means of a web-based user interface. Such a deployable workflow bundle comprises: (a) a executable workflow where service tasks have been mapped to certain services/SaaS; (b) a deployment plan which indicates where (in the cloud) the BPaaS internal components are deployed; (c) monitoring and adaptation information to guide the adaptive provisioning of the BPaaS workflow; (d) SLA specification explicating the exact service level to be offered by the BPaaS.

As soon as a BPaaS bundle is ordered in the Marketplace, the BPaaS Bundle is transferred to the BPaaS Execution Environment. This environment is responsible to manage, monitor and adapt the execution of the BPaaS bundles generated during the allocation phase. The execution comprises the deployment and orchestration of the required cloud services via the Cloud Provider Engine, the preparation of the Workflow Engine to interact with the deployed services and the monitoring of the holistic BPaaS lifecycle. When a BPaaS workflow bundle is deployed, the environment will allow to manage the workflow instances created by the BPaaS Customer and to visualize the conformance levels to associated agreements and respective monitoring data. Besides, based on the monitoring data, the violations incurred as well as the BPaaS bundle adaptation rules, the environment will be able to adapt the BPaaS instances to maintain the promised service level via executing particular adaptation actions, including component scaling, component/workflow migration and service substitution, possibly across different levels (Workflow as a Service (WaaS), SaaS, PaaS, & IaaS).

This Deliverable D3.3 describes the “BPaaS and Allocation Environment Research Blueprints”, including modelling, allocation and execution related challenges and solutions. Based on the resulting blueprints of D3.3, the follow up Deliverable D3.4 “BPaaS Allocation and Execution Environment Prototypes” will analyse the actual blueprint prototypes, i.e., the (almost) mature research prototypes/assets that could be adopted by the CloudSocket implementation.

The concepts/blueprints of D3.3 and prototypes of D3.4 will provide the required input to the upcoming Deliverable D3.5 “BPaaS Monitoring and Evaluation Blueprints” in M24. D3.5 will close the loop of the holistic BPaaS lifecycle with the focus on the Evaluation Environment providing analysis capabilities that result in business intelligence knowledge through KPI analysis and drill-down, SLA violation patterns detection, best BPaaS deployments discovery and determination of optimised billing models for the CloudSocket broker.

1.2 Research Problem

With the definition of the business processes and respective workflows in the Design Environment, the Allocation and Execution Environments enable the deployment and adaptive provisioning of workflows in the cloud. Therefore, the Allocation Environment enables the mapping of abstract workflows to deployable and executable solutions, namely BPaaS Bundles, by receiving semantically enriched models from the Design Environment, which are translated into a technical allocation description.

The identified research challenges are introduced by the means of the business process “Sending Christmas Greeting Cards”, which was introduced in D5.1. This business process requires three different kinds of services, an email service, a CRM service and a card designer service, which need to be mapped to respective allocation decisions constituting an executable business process solution that is deployable in the cloud. In the context of this deliverable we focus on the technical realization and do not reflect the modelling guidelines presented in D5.2 In

order to enhance the BPaaS lifecycle, three blueprints categories are derived from the high-level architecture shown in Figure 1 and the related tasks: BPaaS Modelling Blueprints, Allocation Environment Blueprints and Execution Environment Blueprints. Each blueprint category comprises a set of research assets, which represent the actual prototypes. Figure 2 provides an overview of the identified research challenges that are described in the following paragraphs.

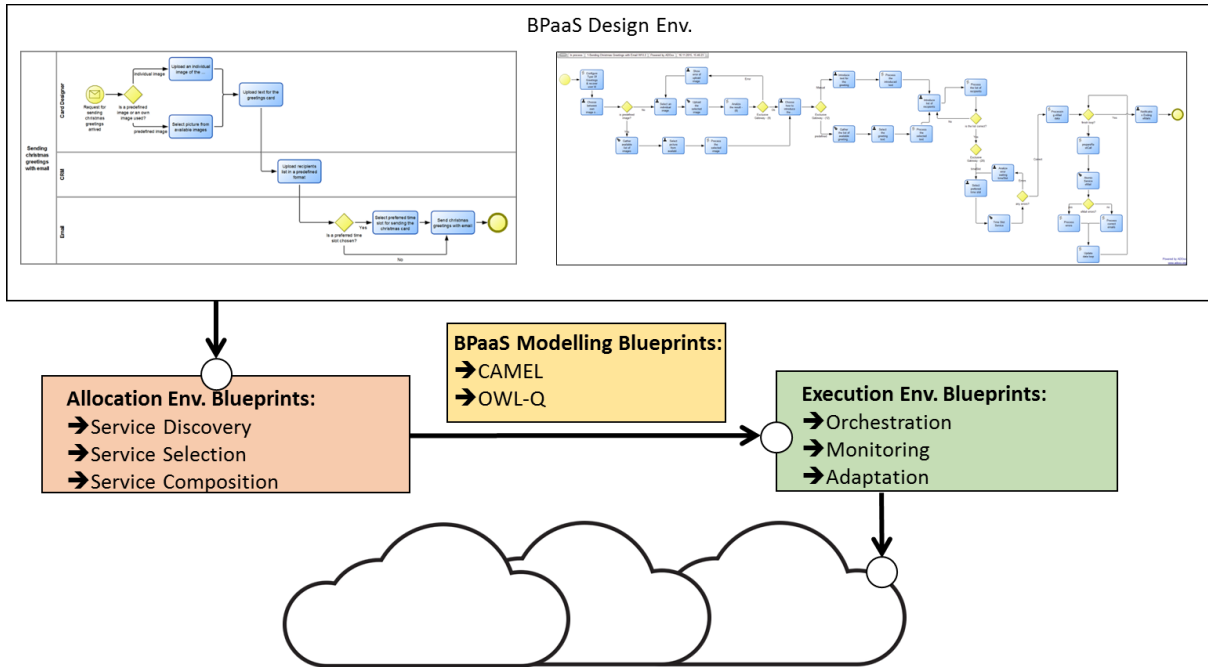


Figure 2 - Identified Research Challenges

The first blueprint, the BPaaS Modelling Blueprint, is represented by the yellow box in Figure 2. The respective research challenge involves the necessity of a smart description for the holistic BPaaS lifecycle, including deployment, adaptation, and monitoring as well as the support of semantics for each aspect. The modelling research area of Domain Specific Languages (DSLs) provides an established set of DSLs: TOSCA [2], CloudML [3] and CAMEL [4]. However, these approaches only target a subset of the aforementioned aspects and none of the existing modelling approaches targets the BPaaS domain explicitly. To provide to the Allocation Environment the complete mapping between business processes and a deployable BPaaS Bundle, the evaluation and extension of appropriate modelling solutions comes up as the first research challenge.

The second blueprint category, the Allocation Environment Blueprint, is depicted in the red box in Figure 2 with respect to the high-level architecture. This blueprint comprises the actual allocation of the BPaaS Bundle while also provides rules for its runtime adaptation. The allocation relies on the use of smart and semantic discovery and composition algorithms which attempt to map BPaaS workflow tasks to concrete cloud services by also respecting the main broker requirements. The respective section (3) will present these algorithms as well as indicate particular research challenges that still need to be satisfied. The broker is able to specify adaptation rules in a high-level language, which is transformed into CAMEL by utilising the Decision Model and Notation (DMN). It can be facilitated by the existence of metric blueprints (i.e., sets of already specified and re-usable metrics) as well as the findings from the Evaluation Environment in terms of event patterns leading to SLO/KPI violations.

The context of the third blueprint, the Execution Environment Blueprint, is shown in the green box in Figure 2. This blueprint is separated into the research assets for orchestration, monitoring and adaptation of the BPaaS Bundle. The holistic lifecycle of a BPaaS Bundle requires the deployment and orchestration of services across all cloud service levels. Whereas recent deployment tools focus solely on the IaaS level, higher-level deployment tools also covering the PaaS or SaaS levels are not yet specifically targeted in academia and industry. All cloud service

levels need to be considered also by the BPaaS monitoring solution. Whereas current monitoring solutions typically focus only on a specific cloud service level, cross-level monitoring is required to cover in a more complete manner problematic situations that impact multiple levels within the BPaaS stack. Monitoring across all service levels also raises new challenges, including scalability to provide a monitoring solution with a suitable performance level. Cross-layer BPaaS adaptation is also a necessity in order to address problematic situations in a holistic manner by also preventing cases where individual level-based actions are performed which are overlapping or conflicting.

1.3 Structure

The structure of this document is organised in the following chapters: chapter 2 introduces the BPaaS Modelling Blueprint, which will be exploited by the Allocation Environment to design the BPaaS Bundle. The BPaaS Modelling Blueprint comprises cloud specific DSLs as well as semantic languages. Chapter 3 describes the actual Allocation Environment Blueprint, including smart service discovery and composition tools and the adoption of DMN to semi-automatically create CAMEL. Chapter 4 presents the Execution Environment Blueprint for the specific sub-phases of BPaaS deployment, monitoring and adaptation. Chapter 5 describes requirements imposed by the identified research items of the previous chapters with respect to their interaction with other CloudSocket Environments. Finally, Chapter 6 concludes with the research showroom, summarising and categorising all research items and their actual research state. Moreover, a brief description of the research handover process to WP4 is provided.

2 BPAAS MODELLING BLUEPRINT

The various tasks that need to be performed for the allocation, execution, monitoring and adaptation of a BPaaS require the existence of models that provide information that properly supports these tasks. A more detailed view on the BPaaS Modelling is provided in Figure 3. The models have to enable the mapping between workflow tasks and actual cloud services, including all technical details to allow the deployment of the service in the cloud. The models should also support the definition of adaptation rules in order to support the adaptive provisioning of BPaaS, which will obviously include the appropriate measurement details required for BPaaS monitoring. In addition, the models need to support the definition of SLAs as well as cost models.

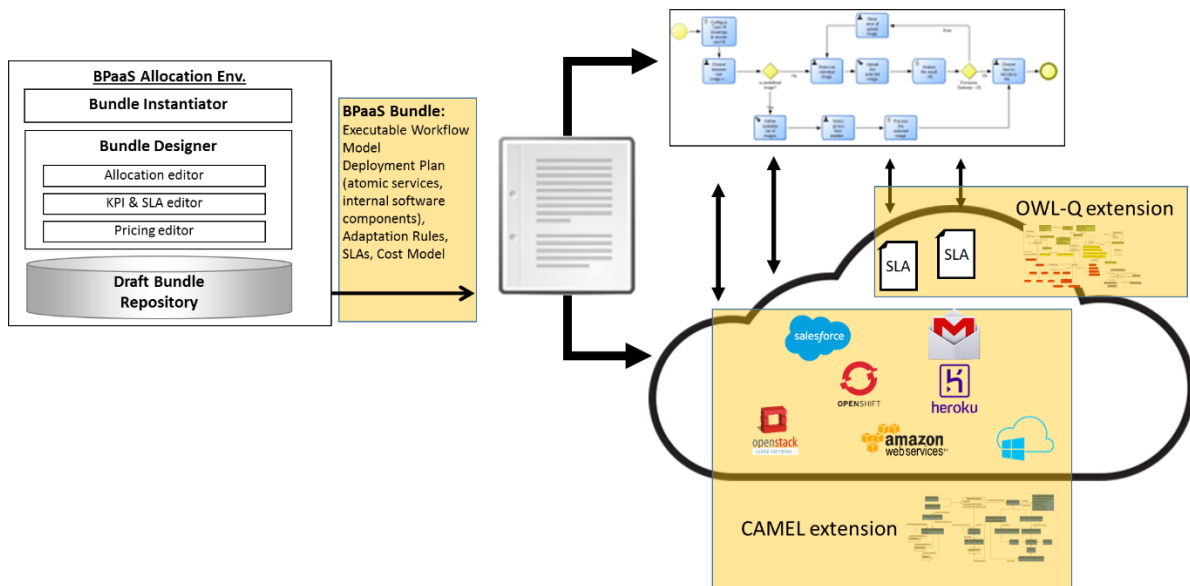


Figure 3 - BPaaS Modelling Blueprint

Such models should conform to one or more languages that define the structure, the main notions and the relationships between these notions. Concerning the BPaaS deployment and adaptation, the language that the consortium has selected is CAMEL [5], a main research result developed in the context of the PaaSage¹ European project. For the semantic modelling of metrics and SLAs, OWL-Q [6] has been chosen due to its ability to express all appropriate aspects related to the modelling of quality terms and service levels. These are the two main languages that are to be used for the research prototype environments.

In the following, after conducting a state-of-the-art analysis concerning the modelling in cloud computing especially spanning the aforementioned lifecycle activities, we describe the main project research contributions concerning the modelling aspect. Figure 3 shows the targeted area for the resulting blueprints. The blueprints are incarnated into the analysis of CAMEL and its main extension blueprints with the focus on the deployment and adaptation of cloud services. An equivalent analysis of OWL-Q is performed, encompassing the respective extensions blueprints with the focus on SLA specification.

2.1 State-of-the-art

The modelling of services in general and for specific domains like cloud computing is an ongoing research area with a large set of existing solutions. As CloudSocket introduces the quite new BPaaS paradigm, the current state of the art on service and cloud modelling is reviewed towards their capabilities and shortcomings for BPaaS.

¹ <http://www.paasage.eu/>

2.1.1 Service Description Languages

The functional specification of services benefits from a plethora of many different languages. Each language focuses on a different functionality aspect. The most commonly used languages are those used to structurally specify the service interface. Such languages are WSDL [7] and WADL [8] that cover SOAP and REST-based services, respectively.

As structural specifications are not information rich, semantic languages have been proposed to close the gap by also raising the level of service discovery accuracy. In this respect, semantic languages like OWL-S [9] and WSMO [10] have been proposed but have not been undertaken due to the shortage in tools able to support the semantic specification of services as well as to the gap between the knowledge of semantic representation and the current expertise of the service modeller. However, such languages have been extensively used in research prototypes with quite significant results and are assorted with collections of semantic specifications, which map to real-world services. Need to mention here that both languages support the specification of the service I/O as well as its behaviour in terms of pre-conditions and effects. OWL-S also enables the description of the abstract interface of the service covering the interactions needed with the service requester.

USDL is a semi-formal language for business and software service description. This language has been recently transformed to a Linked-Data counterpart [2] to become more formal. Moreover, USDL covers also the specification of SLA, quality, security, cost and legal aspects. An approach in [11] was also proposed focusing on integrating USDL with TOSCA to link service selection with deployment such that the cloud application lifecycle is better supported.

A UML based language called SoaML [12] has been proposed to specify Service-Oriented Architectures (SOAs) by defining components and their inter-relationships at the business and service levels. This language, however, mainly focuses on the functional specification of the services and does not deal with the internal orchestration logic.

As composite services comprise more simpler services that have to be coordinated, quite well-known workflow specification languages can be used to express the internal orchestration logic such as WSBPEL [13]. In addition, recently, BPMN [14] was also extended in order to support the specification of service-based workflows. Both these two languages and especially BPEL4WS are used by the majority of service providers in order to specify the internal logic of their services. In addition, semantic annotations have been considered for both languages in order to assist in the concretisation of abstract workflows. D3.1 [15] also shed light on this in terms of semantic lifting and alignment.

The project has decided to follow a structural-based approach to specify services according to a specific JSON schema. This was mainly done for simplicity and current implementation reasons. However, concerning the main research developments and the need for supporting semantic service discovery that reaches higher-levels of accuracy, the main research prototypes addressing smart service discovery adopt OWL-S due to its main capabilities to semantically describe both I/O and behaviour of the service. Moreover, OWL-S is coupled with a semantic service collection, which can be used as a basis for an extended semantic service registry. In case that the internal logic needs to be captured, then BPMN is by de facto adopted by the project. Semantic annotations are entered in this case in the current registry implementation that in conjunction with the CAMEL deployment plan indicate the way the functionality of an abstract BPMN workflow task can be realised by a specific abstract service entry in the registry, which is semantically annotated.

Various non-functional service description languages have been proposed. A quite detailed evaluation on them can be found in [16]. From this evaluation, it becomes apparent that: (a) there are particular features that distinguish one language over the other, including the formalism, the richness, and the complexity; (b) languages can be separated according to the lifecycle activities that they can cover. In this way, languages covering service quality profiles go until the service discovery while languages covering SLAs cover potentially the whole lifecycle; (c) OWL-

Q is the most prominent language from those used to represent quality profiles while for SLAs, there is actually no language that prevails. In fact, a trend has been revealed where different languages are combined together in order to exploit in a complementary way their SLA specification capabilities. Nevertheless, languages from the first type can be extended to cover the second specification type and this will be witnessed in section 2.3.2.2 where we will analyse the extension made towards enabling OWL-Q to specify SLAs. Such an extension leads to an SLA language that surpasses the current state-of-the-art.

2.1.2 Cloud Service Modelling

A de-facto standard for the description of the application deployment that is widely used in research prototypes is TOSCA [2]. TOSCA is an OASIS open standard that defines a description of services and applications, including their components, relationships, dependencies, requirements, and capabilities. It can be described as a technology centred on the application. The objective is to enable portability and automated management across cloud providers regardless of underlying platform or infrastructure. This way, TOSCA expands customer choice, improves reliability, and reduces cost and time-to-value. These characteristics also facilitate the portable and continuous delivery of applications (DevOps) across their entire lifecycle. However, it comes with certain shortcomings related to the non-coverage of the instance level required for dealing with runtime aspects, the lack of cloud/domain-specific constructs and the almost non-coverage of the non-functional aspect. These shortcomings limit the holistic modelling of the BPaaS lifecycle needed which takes into consideration all cloud service levels as well as various types of technical requirements.

CAMEL is a multi-purpose DSL developed in the context of the PaaSage European project. This language is analysed in detail in section 2.2 and has been finally adopted by the CloudSocket project. Two main drawbacks apply to CAMEL: (a) it is quite lengthy covering a great level of details that might not be required in the context of specific tasks; (b) it is semi-formal as it is Ecore-based. However, the first drawback is solved by the modularity of the language such that only specific modules can be used in the context of a specific task. The second drawback can be solved via enabling semantic annotations via using an appropriate and suitable language like OWL-Q (see section 2.4.2).

In [17], a language enabling the semi-formal description of Blueprint Templates is proposed. Such templates cover cloud-offerings at multiple abstraction levels and capture service capability, virtual topology as well as QoS and policy aspects. Apart from being semi-formal, this language does not capture information that is required in all lifecycle activities as done in the case of CAMEL. In addition, it cannot define the quality terms required for quality capability specification of respective service offerings.

Galán et al. [18] have proposed a cloud meta-model that extends OVF² towards covering self-configuration, elasticity and performance monitoring. This meta-model cannot specify component dependencies as well as quality capabilities and requirements.

The service manifest is another OVF extension proposed in [19]. This extension covers placement and allocation constraints, security requirements and performance profiles according to the properties of trust, reputation, eco-efficiency and cost. However, the service manifest covers mainly the IaaS level without the ability to describe component dependencies. In addition, it does not have the ability to model additional quality attributes and metrics related, e.g., to performance as in the case of CAMEL.

mOSAIC [20] is an OWL-based ontology used for semantically annotating semi-formal cloud service descriptions. It covers various aspects, including cloud service requirements and resources, metrics, SLAs, components and

² <https://www.dmtf.org/standards/ovf>

policies. Such an ontology could be exploited in the context of semantically annotating CAMEL as indicated in section 2.4.2 but mainly covering semantic concepts not currently captured by OWL-Q.

By considering the IaaS level, it can be considered that CAMEL is the richest from all languages able to cover all possible aspects in an appropriate level of detail. In addition, it caters for the models@runtime [21] approach enabling a system to always keep up with an up-to-date state of the multi-cloud applications that are being provisioned. Moreover, it is also coupled with tools that enable both the deployment description, provisioning and local as well as global adaptation of multi-cloud applications.

2.2 CAMEL

In order to cover appropriately all the information aspects involved in the design and adaptive provisioning of multi-cloud application, the PaaSage European project has developed the CAMEL family of DSL languages. CAMEL comprises DSLs which were already existing, such as CloudML [3] and Saloon feature meta-model [22], as well as languages that were developed from scratch in that project, such as the ScalabilityRule Language (SRL) [4]. The information aspects that CAMEL spans include application deployment, monitoring, scaling, cloud provider offerings, organisation, security, as well as requirements modelling.

All meta-models in CAMEL have been specified in EMF³ core. This enables using various technologies provided by the Eclipse framework, including editors and programmatic interfaces. These meta-models were also carefully integrated by removing duplicate concepts or relationships and connecting appropriately related concepts from different aspects/meta-models. This integration is supported via the specification of OCL⁴ constraints that enable proper semantic validation of models in one or across domains. In this way, the modeller is guided in providing only semantically and structurally valid models conforming to the CAMEL meta-model. This guidance is supported both in an interactive mode via editors as well as in programmatic mode. Three editors can be mainly exploited in order to specify CAMEL models. The first one is provided by default by the Eclipse IDE and enables a tree-based editing of the models. The rest of the editors have been developed in the context of PaaSage. The first is a textual editor that conforms to the textual syntax of CAMEL, which was defined by exploiting Eclipse's XText⁵ technology. This editor provides some added-value features like auto-completion, error marking and automatic transformation of the model into an XMI form. The web-based editor has been developed again via Eclipse Technologies (RAP⁶) and enables the web-based editing of CAMEL models focusing more on (deployment and application) requirements specification as well as in the specification of organisation models and especially their security-oriented aspects (user and permission modelling). Its main advantages are that it does not require from the user to know the textual or normal syntax of CAMEL, it immediately generates valid models that are persistent in a corresponding model repository and it enables a role-based access only to CAMEL aspect-specific models which conform to this role's allowed permissions.

Due to this extensive aspect coverage at the more technical level and especially its prominent capabilities to describe both abstract and concrete deployment plans as well as scalability rules, CAMEL was selected as the main cloud modelling language for the CloudSocket project. However, while this language can be used more or less as it is, there are particular aspects that need to be slightly or more heavily extended in order to better support the aforementioned BPaaS lifecycle tasks/operations across all possible layers (IaaS, PaaS, SaaS and WaaS). In this respect, we first provide an overview of the original version of CAMEL and then we analyse in detail the extensions that were performed on it.

³ <https://eclipse.org/modeling/emf/>

⁴ <https://wiki.eclipse.org/OCL>

⁵ <https://eclipse.org/Xtext/>

⁶ <http://eclipse.org/rap/>

2.2.1 Original Version

Our analysis on the original CAMEL version focuses mainly on those aspects or sub-DSLs that are mainly used by the project. These sub-DSLs include the deployment, metric, scaling, requirement, provider and organisation meta-models. These meta-models are going to be shortly described only. More details can be found in CAMEL's technical documentation⁷.

Camel Meta-model. This is the top-level meta-model which defines a root CAMEL model that can encompass aspect specific models. It also enables the definition of applications comprising their name, short description and version. As such, one CAMEL model can be associated to one or more applications for which the respective aspect-specific models should hold.

Deployment Meta-model. This meta-model enables the specification of both abstract and concrete deployment plans. Abstract plans define the structure of the user application in a provider-independent way. They indicate what are the main application components and how they can be configured via respective OS commands, the VM nodes on which these components can be hosted along with the respective requirements on VM characteristics like the number of cores, as well as hosting and communication relationships between the application components. Concrete deployment plans, on the other hand, model the deployment of an application in a cloud provider specific way by working mainly at the instance level. In this sense, the deployment meta-model suitably covers the type-instance pattern. In such plans, each application component or VM maps to one or more instances that are connected to each other according to the respective relationships defined at the type level. Moreover, both application components and especially the VMs that host them map to real IPs. We should also note here that at the instance level we also make a connection between a VM instance and the respective VM offering of a cloud provider that is instantiated (specified in a respective (cloud) provider model). This is essential information for deployment as the respective deployment engine will then know which VM to instantiate at which cloud.

Requirement meta-model: This meta-model enables the specification of various types of requirements. First, requirements can be categorised into hard and soft. Hard requirements must be satisfied at all means while soft requirements are usually optimisation directives over non-functional parameters to the platform over how the best deployment plan can be derived. Hard requirements can be further categorised into hardware, OS, provider, location, and service level objectives (SLOs). The first 4 sub-classes can be associated either to the whole deployment plan, as global requirements that must hold for all VMs, or to specific VMs as local requirements in order to restrain the cloud provider space. Hardware requirements mainly impose restrictions on the values of VM characteristics, which include the number of cores, the memory size and the disk storage size. OS requirements explicate the OS that must be supported by the VM. Provider requirements indicate a specific provider from which respective VM offerings should only be considered to instantiate a specific VM. Location requirements are used to restrain the cloud provider space to a specific location that can be physical or virtual. Physical locations map to specific continents or countries while virtual locations map to locations that are specific for a certain cloud. Finally, SLOs are hard requirements on the application service level, which indicate that the values of a particular quality term (attribute or metric) should not overpass a specific low or upper threshold. Such requirements are mainly used in order to filter the provider space during deployment plan reasoning.

Metric meta-model. This meta-model specifies all necessary measurement details in order to measure specific properties of components at different levels of abstraction. Such measurement details are encompassed in the notion of a metric. Metrics can be raw or composite. A raw metric can be immediately measured via sensors. A composite metric can be measured by applying a specific formula over measurements of other metrics. Formulas are actually expressions that can encompass the application of mathematical or statistical operators over metrics,

7

https://tuleap.ow2.org/plugins/git/paasage/camel?p=camel.git&a=blob_plain&h=62d67508d3611f64d67a88ead10afeef350f711e&f=documents/CAMELDocumentation.pdf&noheader=1

attributes or other (sub-)formulas. In order to cater for scheduling/timing aspects, metrics can be associated to a metric context which indicates when (how frequently) each metric should be computed and according to which measurements (time or measurement or mixed-type of windows). Such a context also indicates other details like which is the object that is being measured (the application or one of its components or a VM) and pattern-based information (pertaining to how many instances of a component should be considered in order to deem a metric condition as being violated or not). Metrics or attributes are associated to conditions that apply a specific threshold on their values. Such conditions are used as building blocks in order to specify scaling rules and service level objectives. As can also be easily understood, metric conditions are also related to a specific metric context in order to set-up exactly the appropriate information to be used for their proper evaluation.

Scalability meta-model. This meta-model can be exploited in order to specify scalability rules. Such rules map a particular event to one or more actions. Currently, horizontal scaling actions are supported as well as event creation actions. The latter lead to the creation of an event when local/cloud-specific scaling fails indicating that global adaptation should be performed for the whole application. Horizontal scaling actions indicate important details about how scaling should be performed by explicating how many instances to create or destroy for which particular application component. Events can be simple or composite. Simple events map to the violation of a metric or attribute condition. Composite events map to unary or binary event patterns. Event patterns combine one or more events according to logic-based (e.g., AND/OR) and time-based operators (e.g., PRECEDES). For instance, we can indicate that we should wait for 10 seconds before a specific event happens or that two different events need to occur in order to consider that the respective event pattern is satisfied.

Provider meta-model. This meta-model has the main goal to specify feature models, which cover all types of offerings of a specific cloud provider. A feature model comprises a tree-based hierarchy of features. Each feature can have a set of attributes whose values map to a specific value type. A feature model is also associated to a set of constraints that can be intra- or inter-feature-based. Intra-feature constraints indicate that e.g. one attribute value of a feature leads to another value for another attribute to be fixed. This is an essential mechanism to specify a mapping from a VM flavour name to the characteristics of this flavour (where VM flavour is an equivalent term to a VM offering or a VM type). Inter-feature constraints can operate over the attributes of the features or their cardinality. For instance, we can express that a particular VM flavour of a provider (mapping to a VM feature) is available only in specific locations (mapping to a location feature). As it can be easily derived, this meta-model is quite generic and can express any kind of cloud provider offering, including IaaS and PaaS services. In this respect, this meta-model does not need to be extended in order to cover all layers in the cloud computing stack.

Organisation meta-model. This meta-model originates from the CERIF standard [112] that is used to specify research organisation information, covering aspects like publications, equipments, users, and roles. A particular sub-set of CERIF was selected as a base for the organisation meta-model and especially concerning the information about user and role modelling. The basic root construct is the organisation model that represents information about a specific organisation, like name, address and web site URL. Specific types of organisation can also be modelled mapping to cloud providers. In this case, additional information can be modelled, like what type of cloud is offered and what of cloud offerings. An organisation model also includes the specification of one or more users. Each user is related to specific personal information, like username, first and last name, and email as well as to credential-based information. Credentials can be platform-specific or cloud-specific. Platform-specific are credentials (in the form of a password) used to connect the user to the respective platform prototype, like the PaaSage (or CloudSocket) prototype. On the other hand, cloud credentials (which can take different forms) are specific to one cloud and represent security information that can be exploited in order to perform cloud-specific tasks (like VM instantiation) on behalf of the user. Users are related to one or more roles. In PaaSage, the basic roles of administrator, DevOps and business have been identified. Each role is associated in turn to a set of permissions that are allowed for it in terms of different types of accesses on platform resources (models or services). Role assignments as well as permissions are associated to specific information that can be used to inspect their validity, like the end date of the assignment or permission.

In the context of the CloudSocket project, the organisation meta-model is exploited by the initial version of the project prototype. However, when the Identity Management solution is ready (to be provided by YMENS for user access control), different formalisms and a different corresponding language might be exploited to specify the same type of information.

2.2.2 Extension: SaaS Modelling

Currently, concerning deployment/allocation, CAMEL only covers allocation decisions for the hosting of application components to VMs. However, the main requirements originating from the project require: (a) allocation decisions to cover additional layers in the cloud computing stack; (b) the respective lifecycles of the components at these layers to be properly handled.

As a first extension to CAMEL attempting to satisfy the above requirements, the deployment meta-model in CAMEL was expanded towards the specification of allocation decisions related to the WfaaS and SaaS layers as well as the coverage of SaaS and internal service components. In the sequel, we explain in detail these two types of extensions.

In order to prepare for the forthcoming extensions towards covering the modelling of PaaS components, a *Component* in the deployment meta-model is now further classified into internal and external components. Internal are software components of the application or BPaaS workflow at hand. On the other hand, external are components mapping to SaaS, PaaS (not currently modelled) and IaaS services (i.e., mapping to VM concept in the meta-model).

A SaaS service is related to: (a) the set of BPaaS workflow tasks for which it can realise the respective functionality (mapping to an attribute of type `String` that can take multiple values mapping to the IDs of these tasks) and (b) a particular ID mapping to the respective entry in the atomic service registry in the main CloudSocket prototype. Such an ID can be a normal identifier or a URI (both can be used as the type of the respective attribute is just a `String`). In this sense, any kind of service repository can be catered from which we can identify and obtain information for the particular SaaS at hand. Please consider that this is usually an abstract SaaS and not a concrete one. This means that it maps to an abstract functionality that needs to be concretised at the instance level. However, we also do allow specifying a concrete SaaS in order to cover all possible scenarios in CloudSocket concerning the concretisation of abstract BPaaS workflows. Such scenarios can be static which indicates that a concrete SaaS has already been identified or more dynamic where we just indicate an abstract functionality and then indicate at allocation specification time how this functionality can be concretised either in a vague or quite specific way. Vague means that we specify a concrete SaaS but do not explicate which endpoint from those available (mapping to its instances) will be exploited; specific means that we also choose the particular endpoint to be exploited.

Covering the case of internal service components, we have also modelled the *InternalServiceComponent* class, which is a subclass of *InternalComponent*. This indicates that an internal service component is a kind of software component that also inherits the respective information pertaining to software components like the ways to configure them. As such, the lifecycle of service components is also handled in a uniform manner as in the case of any other software component. Of course, some differentiation can exist which actually concerns the instance level in terms of issuing different commands in order to handle the lifecycle with respect to other types of components, like databases. In this class, we have also modelled specific additional information which relates to the type of the service (SOAP or REST), mapping to a member of a new enumeration called *ServiceType*, plus allocation information equivalently specified as in the case of SaaS (IDs of tasks for which the functionality can be realised).

To cater for the instance level, the same classification (as in the type level) has been enforced. This means that component instances are further classified into internal and external component instances. The external component instances now map to SaaS and VM instances. A *SaaSInstance* is mapped to a set of endpoints, which indicate the different instances of a SaaS that are available in order to realise the functionality of a BPaaS workflow task. In

case that one endpoint is provided, this means that the allocation decision is fixed for the respective abstract BPaaS workflow task(s). In case of multiple endpoints, the allocation is not actually fixed and has been determined at deployment time when additional information is available. For instance, as each endpoint is location specific, it might be the case that we need to know the location of the BPaaS customer in order to select the SaaS endpoint that is closer to this customer.

Similarly, the `InternalServiceComponentInstance` class has been realised which just indicates the endpoint from which this instance is available. This endpoint is to be set when the respective deployment plan is executed and especially when the respective instance is deployed on a particular VM. We need to mention here that multiple instances can map to the same internal service component. This can occur when we need to split the tasks horizontally into partitions that are covered by different instances for load balancing reasons. To this end, each internal service component instance can be related to a subset of the tasks that are associated to its type. If this information is not provided, this means that the instance realises all the tasks allocated to its type.

We should note here that following the design principles of CAMEL, we have made extensions that are compatible with the original CAMEL version in order to allow easily CAMEL to evolve without requiring modifying existing code of respective platforms. Moreover, we have created an initial small set of OCL constraints, which further enhance the semantics of the deployment domain according to the extensions that have been performed.

2.2.3 Extension: Cross-Layer Description of Components

In the current CAMEL meta model, a component can have multiple configurations. Any configuration is a set of life cycle actions represented as Strings. Unfortunately, the configurations are not annotated with any semantic information about how to use and execute the life-cycle actions on which platform. Obviously, this hinders the execution of a component across multiple cloud service levels.

In the following, we present an approach to develop a method specifying components in a way that will lead to the ability to deploy a service on cloud providers of different service levels and capabilities (IaaS and PaaS).

2.2.3.1 Configuration per service level and ability

Configurations of a component can have different formulations based on the service level(s) supported by the Cloud provider and the ability in terms of DevOps tools to manage the deployment.

On **IaaS** level, a typical approach is to define the life-cycle actions directly on the operating system (OS) in terms of scripts. These are then executed and their correctness is evaluated via environment variables or return values. This approach suffers from an OS-dependent description as well as an imprecise error handling when executing the life-cycle actions.

On **PaaS** level, there exists a numerous amount of provider-specific APIs, focusing on specifying the application and its environment, which are managed by the PaaS provider. There are also APIs to build an abstraction layer to plethora of PaaS providers, such as the COAPS API (see section 4.1.1.2), by defining a spanning format for the application manifest and the environment with its capabilities.

There are also approaches between those levels that make use of IaaS providers and build up the environment for an application by defining its platform in terms **DevOps** tools like Chef⁸ or Puppet⁹. Examples for this are Scalr¹⁰ or Foreman¹¹, which allow to describe nodes of your deployment by means of Chef recipes or Puppet modules.

⁸ <https://www.chef.io/>

⁹ <https://puppet.com/>

¹⁰ <http://www.scalr.com/>

¹¹ <https://theforeman.org/>

For each service level and DevOps ability, there are multitude of variants to consider before moving from one Cloud provider to the other.

2.2.3.2 Overloading the component configuration

The first approach to solve this on the modelling side, is to allow the application engineer to specify multiple configurations for each service level and DevOps ability. Here, we present an extension to CAMEL, which only needs few adjustments to the language itself.

Figure 4 shows the class structure for the *Configuration* entity referred from the *Component* entity, which is needed in this solution. It shows that the *Component* entity can now have attached several different kinds of *Configuration* entities. In particular, a *Configuration* is specialized in a *ScriptConfiguration*, *DevOpsConfiguration* and *PaaSConfiguration*. The *ScriptConfiguration* is expanded by life-cycle actions that are already supported by the Cloudiator toolset, while is almost identical to the original *Configuration* class in CAMEL. *DevOpsConfiguration* will be specialized to several DevOps tools, such as Chef or Puppet. In *PaaSConfiguration*, it will be possible to specialise for different PaaS providers (e.g., Heroku) but also cater for a cross-provider description, as is the case of e.g. the COAPS API manifest for environments and applications (see section 4.1.1.2).

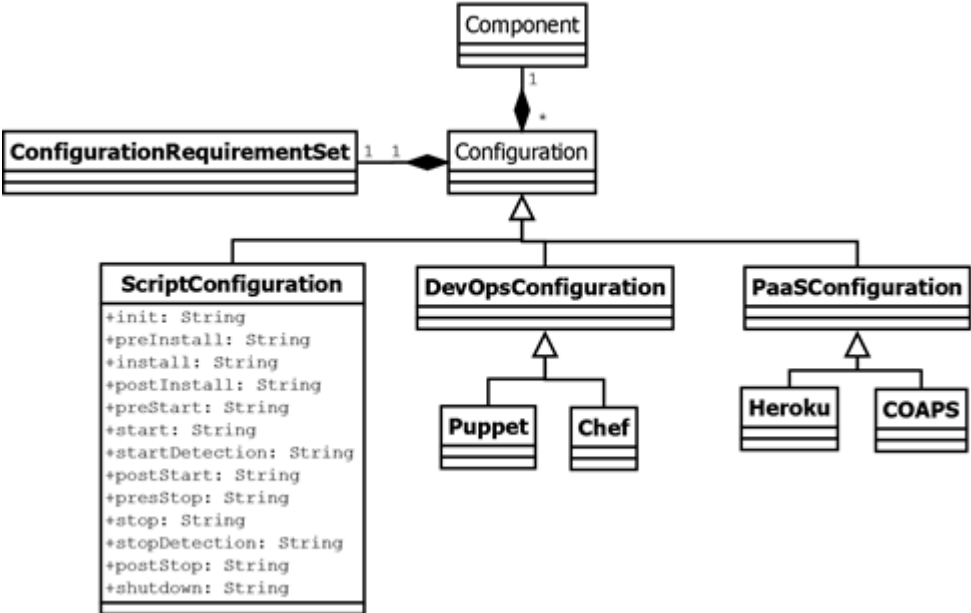


Figure 4 - Advanced class structure of the configurations of a component in CAMEL

A *Configuration* has now also attached a *ConfigurationsRequirementSet* whose structure is visualised in Figure 5. The semantics of this entity is based on the *VmRequirementSet* of the current CAMEL version. This means that it comprises several entities that describe the requirements for this *Configuration*. The updated *ProviderRequirement* entity defines restrictions of the Cloud Provider for this configuration, e.g., it has to be an IaaS or PaaS provider, or support a certain PaaS API or PaaS abilities, such as storage- or messaging-specific capabilities. For the latter restriction, the current CAMEL has to be extended by additional configuration values in this class, which are represented by Strings in the field's type, name and version. As with the current CAMEL version, it is still possible to define specific Cloud Providers as a requirement. A *HardwareRequirement*, has not been modified with respect to the current CAMEL version, and defines hardware constraints such as the number of CPU cores or size of RAM. An *OsOrImageRequirement* imposes the usage of a certain image or operating system. A *SoftwareRequirement* defines which software has to be made available for the *Component* to run with. By that, it will be possible to define a *ScriptConfiguration* that has no *SoftwareRequirement* defined, when the scripts actually install the software, besides a *PaaSConfiguration* that describes the PaaS capabilities in terms of *SoftwareRequirements* that are needed in order to run a certain component.

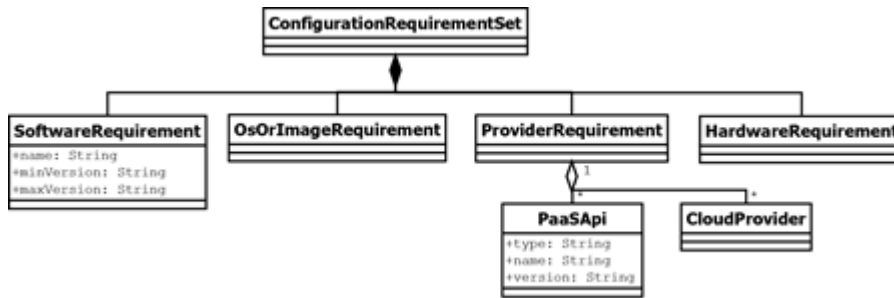


Figure 5 – The ConfigurationsRequirementSet part of the class structure

The life cycle that is managed per *Component* is directly defined in the *ScriptConfiguration* as actions represented as Strings, whereas the management of life-cycle actions of *DevOpsConfiguration* defined by the actual DevOps tool. *PaaSConfigurations* have their life cycle pre-defined by the provider or an abstraction layer. In the latter case, the *Configuration* maps to the parameters of the predefined life-cycle actions of the PaaS providers, such as "createEnvironment", "createApplication", and "destroyEnvironment".

2.2.3.3 Crushed configurations by building blocks

Another way to realize this multi-definition of the life-cycle actions is to rely on the principle of building blocks from which the user plugs his application together and for each block, a description for multiple cloud-service levels and DevOps tools is available. This is a more sophisticated approach as we not only add additional configurations for the same topology, but also integrate the concept of *Containerizables* and *Containers* (cf. Figure 6). A code, that is an actual component of an application, can be a *Container* or *Atom*. We define *Atoms* as atomic services that are pre-configured and ready-to-use software (delivered as SaaS), which is not yet in the focus of this configuration model. A *Container* is a building block that provides a hosting platform for a *Containerizable*. A *Containerizable* is a template class, which is related to the *Container* it can be hosted in. A *Container* can host severable *Containerizables*. This allows the user to plug the building blocks together that he/she needs.

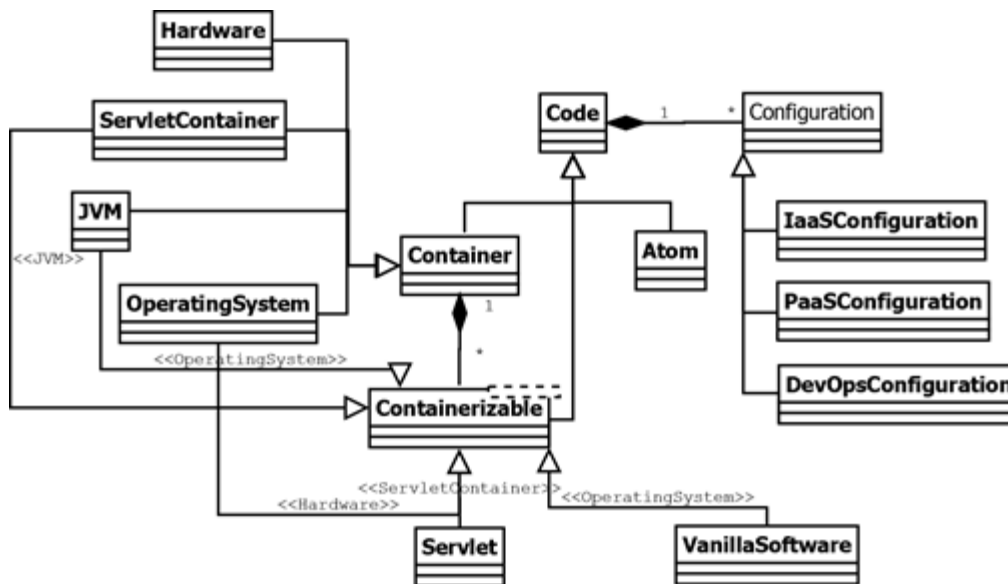


Figure 6 - Class structure for Container and Containerizables concept

A Common specialization for *Containers* is *Hardware* associated with a description about the essential hardware capabilities as CPU, RAM and Storage. An *OperatingSystem* is a *Container* as well as a *Containerizable* for *Hardware* Containers. A *JVM* is a *Container* as well as a *Containerizable* for an *OperatingSystem*. The OpenJDK for example is a specialized *JVM*. For the *Provider*, as seen in Figure 7, we need to define explicit entities for *Containers* and *Containerizables*, so the user is able to choose or define them on his/her own, which would then

mean to provide also the *Configuration* entities. This way, every *Code* (that can be an *Atom*, *Container* or *Containerizable*) has multiple *Configurations* attached, which aim for different Cloud-service levels. *IaaSConfiguration* remains with executable scripts and a reference to an Operating System. A *PaaSConfiguration* contains the part of the application manifest that deals with this part of the component of an application. Any *SaaSConfiguration* is to be specialized for each SaaS provider. *SaaSConfiguration* includes provider-specific mechanisms for, e.g., accounting and provisioning that is out of focus of the current research. By this structure, if a user uses for his *VanillaSoftware* the *OperatingSystem Container Ubuntu*, which is a parent class of *Ubuntu_14_04*, the Allocation Environment will be able to use any sub-class of this *Container*, which is provided from any of Cloud provider – implied that no other restriction is given in the deployment rules.

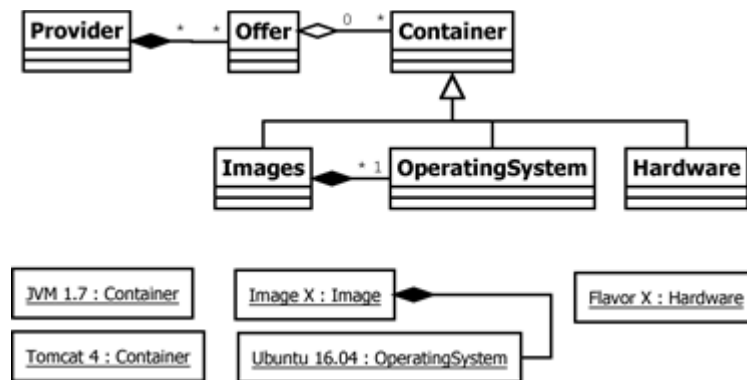


Figure 7 - Provider model for Containers and example instances

This approach caters for an easy-to-reuse container model by the concept of building blocks. Figure 8 shows an example stack of *Containers* and a derivation of a *Servlet*, which implements the actual life-cycle actions (e.g. in case of an application manifest) in the context of a user-defined custom application component.

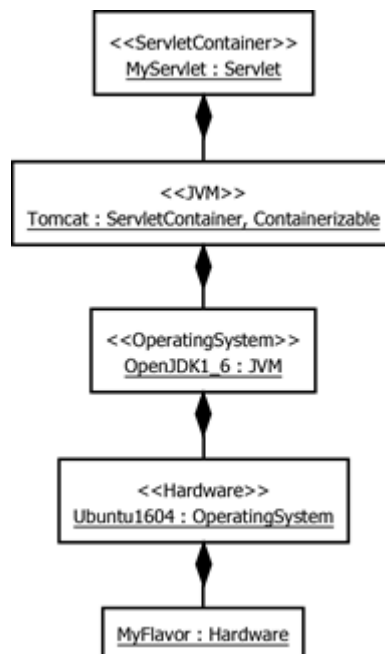


Figure 8 - Example for a application stack from building blocks

In the end, the proposed modelling might serve as a meta language for CAMEL, since the Containerization capabilities will be mappable to the current *Hosting* concept, as it will then be able for multiple components to share the same hosting capabilities. In addition, the very generic provider model of the current CAMEL version needs to be specialized by enriching it with the ability to define Components with hosting capabilities.

Due to the immense genericity of CAMEL's provider meta-model, a mapping to this meta-model is possible, but semantic definitions on how to use attributes and constraints, need to be made. Figure 9 shows a simplified example of how such a language construct could look like. This of course requires to have entities like Tomcat or JVM already defined, such that the user does not have to care for the complex Provider model needed to map the application deployment description to actual available offers. For such entities, we need an ontology or repository, such that the same terminology is adopted in defining both, the cloud service requirements and cloud service capabilities. This is up to be integrated in the modelling/development environment, such as the user just needs to provide the minimum possible information to define his/her application.

```

Application "Example":
  MyServlet [quantity: 1- 10] /* '[' denotes further configuration and
                                set of requirements */
  Loadbalancer [quantity: 1]
    -> MyServlet /* '->' denotes communication */

Component "MyServlet":
  Servlet [/* custom manifest and custom
            life-cycle action scripts,
            means here IaaS and PaaS
            deployment is possible */]
  Tomcat [version: [3.1; 3.3-40]]
  JVM [version: [7-8]]
  OperatingSystem [LINUX]
  Hardware [CPU: [1-4]]

Component "Loadbalancer":
  Nginx [/* custom life-cycle action scripts, means here only IaaS */]
  Image [Image_X]
  Hardware [CPU: [1-4]; RAM: [4-8]]

```

Figure 9 - Simplified application description

2.2.3.4 Conclusion

For CloudSocket we aim to be maximally flexible in choosing the right Cloud provider. This will enable us to be more cost-efficient by, e.g., using a cheaper hosted platform from a PaaS provider, instead of creating the platform on our own based on virtual machines. Other criteria for provider selection are, e.g., trust and availability.

The first approach affords the least effort for extending CAMEL and integrating it into the current system. However, this approach would involve duplicate definitions of component configurations and would therefore most likely not be feasible as a direct interface for, e.g., DevOps engineers. As this is not the main target group for CloudSocket, we would most likely go this very first alternative approach and then widen the user interface capabilities.

Using building blocks as highlighted in the second approach, will help the modeller to (re-)use the same artefacts across multiple applications. The idea of CloudSocket's architecture relying on building blocks, that can be independently used and exchanged, is reflected in the latter approach and will make it easier to be used by DevOps engineers, by having a way to plug their components together.

In order to make CAMEL more usable for actual human users, a meta-language approach is feasible as described in the second approach. However, this would also demand a high amount of effort as this approach implies many changes to the CAMEL meta-model. The first presented approach aims on fulfilling the main goals of a cross-layer description of components, i.e., the ability to deploy the same component across multiple provider (types) and technically map and integrate common DevOps approaches, by as few modifications to CAMEL as possible. At the current stage of CloudSocket, DevOps are not the main target user group of the project. In addition, CAMEL is not used as a direct interface for the user, but created in a semi-automatical way. Evaluating these priorities and characteristics, we suggest going for the first approach. The effort for huge modifications to cater for DevOps engineers does not give much of a benefit to the project at the current stage.

2.3 OWL-Q

OWL-Q [6] is a prominent semantic service specification language that covers completely the non-functional aspect by allowing the description of quality models as well as quality specifications. It has been carefully designed into different facts, which cover various aspects in the definition of non-functional terms and specifications. Quality models are specifications of quality terms, like quality metrics, attributes and groups, which cover the relationships between these terms. For example, a quality model can specify the group of *performance*, which can include the quality attribute of *response time* as well as the *mean response time* metric able to measure this attribute. Such quality models can be regarded as standardised vocabularies covering domain-independent and domain-specific quality terms that can be used in the specification of quality capabilities or requirements in quality specifications.

Based on the survey in [16], OWL-Q has been considered as the most prominent service quality description language which has the major drawback of being quite lengthy. In addition, by considering the context of this project, this language needs to be extended in order to cover: (a) the consideration of components in all possible layers; (b) the specification of SLAs as a special kind of non-functional service specifications. Through satisfying the requirement in (b), OWL-Q can be considered to apply across the whole service lifecycle and not just the service description and discovery activities. This is the main advantage of having the ability to specify SLAs. However, this also must be properly supported through modelling all the necessary information to support the remaining lifecycle activities. As such, OWL-Q has been updated in order to alleviate all these aforementioned issues. In the sequel, we explain shortly what was the original version of OWL-Q and then we analyse the main extensions performed on it.

Before explicating OWL-Q and its main extensions, we need to highlight a specific additional issue that might be raised by the conscious reader. Both OWL-Q and CAMEL seem to overlap with respect to quality term specification. This is indeed true but the usage of these languages in the context of this project will be complementary. OWL-Q will be mainly used for specifying the semantics of quality terms mapping to a vocabulary of terms that can be exploited in order to specify monitoring conditions and event patterns leading to the firing of scalability rules via CAMEL. In this sense, OWL-Q semantically annotates CAMEL and enables the lifting of the monitored information. In this way, such semantically-lifted information can then be exploited for the evaluation of KPIs as the Evaluation Environment employs a more semantic approach for supporting this and other types of analysis.

To add to the above discussion, we should also mention that the goal of WP3 is to research and develop interesting research prototypes, which could be exploited by the implementation of the CloudSocket prototype in WP4. In this sense, OWL-Q can be used in specific research prototypes covering the monitoring of BPaaS or their components (see section 4.2.4), thus completely substituting CAMEL in this lifecycle activity. To this end, the synergy between OWL-Q and CAMEL will be surely exhibited in the main CloudSocket prototype while in WP3 we have the freedom to use different types of languages in order to support different types of BPaaS lifecycle activities. This is also evident from the fact that OWL-Q is proposed to cover SLAs in WP3 while in WP4 WS-Agreement is used for this coverage due to the use of the SLA Manager from ATOS.

2.3.1 Original Version

OWL-Q originally comprised eight facets. In addition, its design evolved around the specification of detailed class hierarchies in order to cover all possible sub-types of the basic quality term types. This design also included the specification of semantic rules in order to capture the respective domain semantics by supporting semantic model validation and derivation of added-value knowledge. In the sequel, we shortly analyse each facet in order to understand the main information aspects covered by OWL-Q. More details can be found in [23].

Connecting Facet. This facet had the main goal to connect a quality profile, whether it covers non-functional requirements or capabilities, into the respective service for which it applies. Such a connection regarded that OWL-S is used for the semantic specification of the service functional part. We should highlight here that via this

Copyright © 2016 UULM and other members of the CloudSocket Consortium
www.cloudsocket.eu

connection we were able to relate one service with many quality profiles mapping to the different levels of performance that this service could support catering for different types or classes of consumers. This facet also covered the specification of different types of quality attributes.

Core Facet. This facet specified the main notions and their relationships for quality specification. It covered mainly generic notions with some of them being further elaborated in aspect-specific facets. Some generic properties were also covered like names and descriptions for any kind of notion. This facet also covered the modelling of quality specifications. Such specifications were categorised into quality offers and demands mapping to the description of quality requirements and capabilities, respectively. Quality demands were actually included in quality requests which also encompassed the specification of quality selection elements (i.e., preferences over quality term specified by the requester). Any quality specification was mapped to a set of (simple) constraints that were expressed as comparison expressions over one or two arguments (where argument was considered as a metric, attribute or metric formula).

Metric Facet. This facet was used to specify metrics, which represent the main notions that encapsulate all measurement details needed for the monitoring of quality attributes. Similar categorisations with respect to CAMEL metric meta-model applied here which were however more sophisticated. For instance, metrics could be classified as either dynamic or static or as either positively or negatively monotonic. Compared to the CAMEL metric meta-model, the notion of metric context was not totally covered and only in an indirect manner. Finally, we should note that metric computation formulas in OWL-Q could be specified either explicitly via respective direct language constructs or indirectly via the specification of such computational expressions in mathematical languages (e.g., OpenMath).

Function Facet. As indicated in the previous sentence, OWL-Q had two ways to specify metric computation formulas. By focusing on the direct expression way, various notions were included which mainly concerned differentiating between how the formula is represented and how it can be applied to a specific metric. Both mathematical and statistical operators were modelled and could thus be exploited.

Measurement Directive Facet. This facet included the modelling of necessary details for specifying measurement directives to be exploited for obtaining measurements for raw metrics. Specific types of measurement directives were modelled for this reason, like gauges and counts.

Schedule Facet. This facet included the modelling of schedules which focused mainly on the frequency of measurement for metrics. Window-based information was not properly covered.

Unit Facet. This facet included the specification of all notions required for the modelling of units. Three different types of units were considered: basic, multiples and derived. A basic unit was associated to a system of units, a multiple was a multiple of a basic unit, while a derived unit was a unit that could be computed from the division of other units. Equivalence of units as a notion was also captured in order to cover mainly the equivalences between basic units in different systems of units. As in the case of the metric computation formulas, equivalence expressions between two units could be expressed in two alternative ways (direct formulas or mathematical expressions).

ValueType Facet. This facet included the specification of value types mapping to metrics. It included a quite extensive classification. First, value types could be classified into *Scalar*, *List-Based* and *NumericUnion*. *Scalar* were further classified into *String* and *Numeric*. *Numeric* could be further distinguished into constrained numeric mapping to range-based numeric types with both or one limit explicitly specified (i.e., mapping to a specific value). *NumericUnion* in turn represented unions of non-overlapping numeric types.

2.3.2 Extensions

Based on the aforementioned main issues, OWL-Q was extended accordingly in order to become compact as well as cover different layers and the specification of SLAs as needed by the CloudSocket project. In the following, we separately analyse the extensions into two main sub-sections covering: (a) OWL-Q necessary modifications not related to a new aspect; (b) the OWL-Q update as a new sub-aspect of the specification aspect to cover the description of SLAs.

2.3.2.1 Core Extensions

OWL-Q was redesigned in order to become more compact. This has led to the production of only 6 facets which resulted in merging some facets and in deleting others. In addition, the design rationale was modified for each facet. In particular, more shallow classifications of concepts were maintained catering for the actual usage of the language and not the coverage of extreme cases. Moreover, while previous OWL-Q version took the approach of enabling users to explicitly state all classes of a particular instance, in the new version one class is usually enough. This is due to the fact that the ontology class axioms have been enriched allowing to infer the rest of the classes on which an instance can belong. Finally, OWL-Q was enriched with a more extensive set of semantic (SWRL) rules covering additional and more complicated validation scenarios as well as the generation of more extensive knowledge facts.

We should also mention that OWL-Q caters for different modes of modelling. The first mode concerns the modelling of anything via the use of sub-classing and more specific ontology class axioms. In this sense, metrics like *Mean Response Time* would be modelled as subclasses of the *Metric* class. This mode can be quite convenient in producing semantic quality models that can be exploited in order to semantically annotate, e.g., the parameters involved in SLO conditions in SLA languages like WS-Agreement. The second mode concerns that the modelling of more concrete things maps to the instance level, i.e., to the instances of the core classes. In this sense, following the same example, *Mean Response Time* would be modelled as an instance of the *Metric* class. The advantage of this type of modelling is that it is more lightweight and can be used to model precisely all appropriate information which can be needed in subsequent lifecycle activities from service/BPaaS discovery. For example, it can capture specific details that can assist in the monitoring of the respective metric specified. A mixed mode could be also supported enabling the instantiation of more specific metric classes and the inheritance of the respective information specified at the class level.

In the sequel, we shortly explain the content of the main facets of OWL-Q. More details can be found [6]. A snapshot of the OWL-Q facet covering all of the facets (apart from the SLA extension) is depicted in Figure 10 and Figure 11.

Central Facet. In comparison to the previous OWL-Q version, this facet has been simplified and a specific part of it was moved to the *Attribute facet*. Now, this facet (with concepts coloured in white in Figure 10) only contains generic notions, relationships between them (like compatibility and dependency) and generic properties (like *value* mapping to a value of specific XSD type). It also includes the specification of quality categories which represent meaningful groups or partitions of other quality terms (attributes, metrics or more specific categories). We should highlight here the notion of an *Argument* (as in previous version) which explicates the different types of arguments that can be used as input in a metric computation formula. As such, an argument can be a quality metric, a quality attribute, a service property, a value (see *ValueType* facet) or a formula (recursive definition).

Attribute Facet. This facet (with concepts coloured with yellow) was also simplified with respect to the previous version of OWL-Q. It now includes only a small shallow classification of quality attributes. In particular, quality attributes can be further distinguished into composite, measurable and unmeasurable. A composite attribute represents a more abstract or complex attribute that can map to simpler ones. For instance, the *response time* attribute can be separated into *execution time* and *network latency* attributes. A measurable attribute can be measured by one or more metrics. On the other hand, an unmeasurable attribute cannot be directly measured. If it

is composite/abstract, this means that probably its descendants can be measured. In some cases, however, such an attribute, if it is concrete enough, can be associated to a specific unit and value type, thus representing a fixed quality characteristic. Any attribute is also associated to the level it maps to. Concerning the case of the extended cloud computing stack (quite relevant also in the context of this project), the following levels are relevant: IaaS, PaaS, SaaS, WaaS and BPaaS.

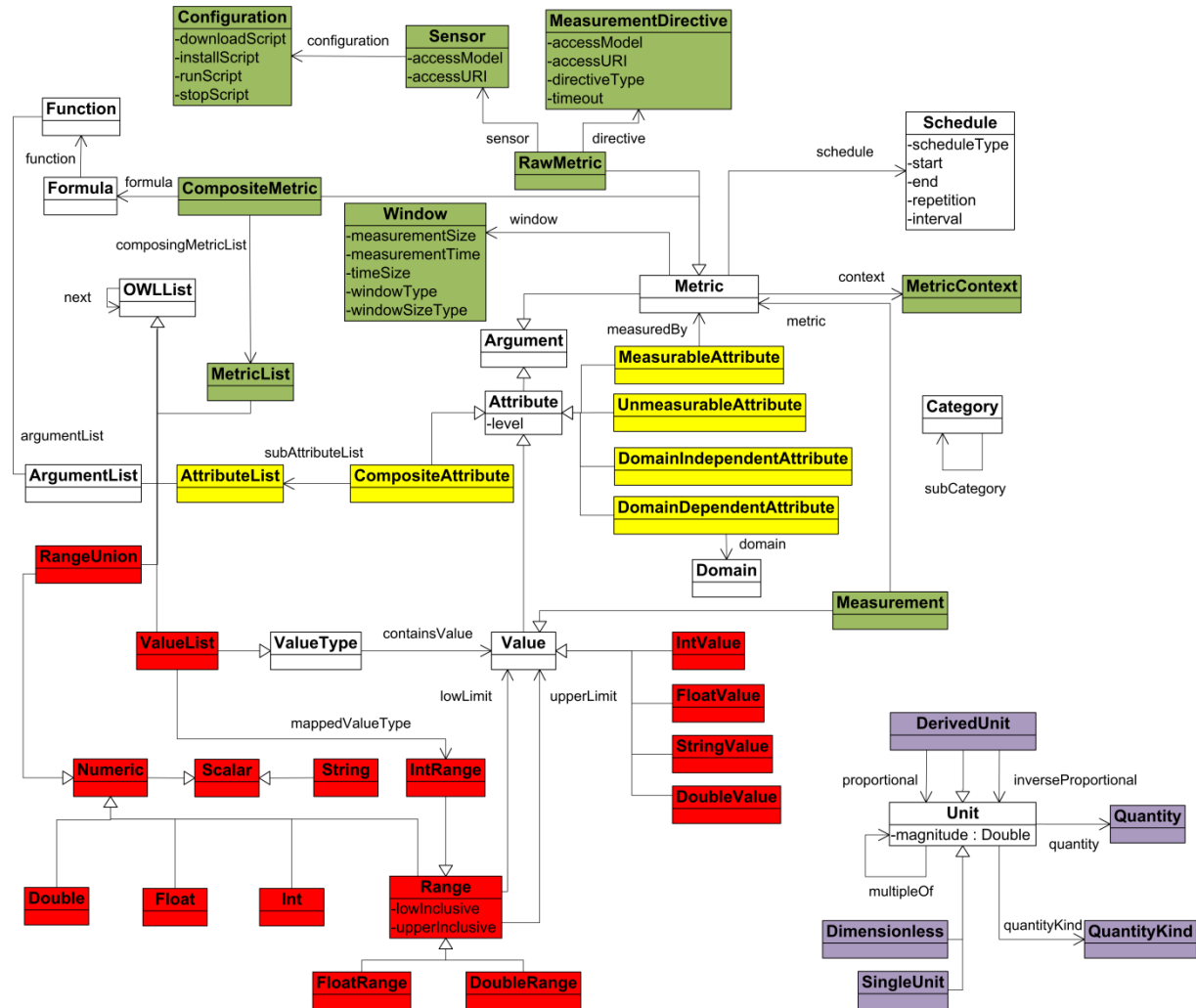


Figure 10 - Five main OWL-Q facets

Metric Facet. Figure 10 depicts in a green colour the concepts of this facet. As in CAMEL, metrics can be considered as raw or composite. Similarly, raw metrics can be computed from sensors or measurement directives, while composite metrics from other metrics via computation formulas. Compared to the previous version of OWL-Q, we now explicitly cover sensors and their configuration and we resort to just one way, the direct one, to express computation formulas. The main rationale is that we allow any kind of interpretation and respective transformation of such formulas into different forms. For instance, such formulas could be translated into SPARQL queries to be posed on a semantic repository over (lower-level) metric measurements in order to dynamically compute the value of a higher-level metric. Formulas are modelled similarly to CAMEL by applying a specific function over an argument list. Functions can be statistical or mathematical and we provide respective classes to represent them. Also the notion of a *MetricContext* has been partially included which was inspired actually by CAMEL. This notion is related to the notion of a metric which enables us to define multiple contexts of the same metric catering for the variability in metric measurement exhibited in monitoring systems. As expected, this metric context covers scheduling and window details concerning the frequency and size of measurements for metric computation. A metric is also associated to a specific unit and value type as well as to a specific monotonicity (covered this time

by a specific attribute). Finally, this facet covers the modelling of metric measurements which are associated to a specific value and timestamp. This allows us to use OWL-Q in order to populate semantic measurement databases which can enable different types of analysis over the measurements modelled and stored.

Unit Facet. This facet (with concepts coloured in purple) was carefully redesigned in order to exclude details not necessary needed from the previous OWL-Q version (like the capturing of units of systems and basic units) as well as to improve some parts of the modelling. In this way, the basic root notion is again *Unit* which is now classified into *Simple*, *Derived* and *Dimensionless*. Simple are atomic units (like *bytes*) which cannot be derived from other units. Derived units (like *bytes per second*) are derived from the division of units multiplications. To this end, two object relations were modelled to cover the nominator and denominator part of this division named as *proportional* and *inverseProportional* as well as a *factor* mapping to a constant of double precision also required for the proper specification of this division. While simple and derived units are associated to a specific dimension (named as *QuantityType*), this is not the case for *Dimensionless* units (like *percentage*). On the other hand, any kind of unit is associated to a specific quantity which is associated to a certain dimension (i.e., type). For example, the dimension of *speed* can have as quantities the network speed (with *bytes per second* as unit) or the light speed. As such, the quantity is the main differentiation or partitioning factor among units of the same dimension.

ValueType Facet. This facet (with concepts coloured in red) can be used to model value types. As the main subclasses of *ValueType* are more or less similar with respect to the previous version of OWL-Q, we focus only on the specific modifications performed. First, the *Range* class has been introduced which always have a lower and a upper limit, where a limit is a kind of *Value*. Specialised instances of values have also been modelled to represent positive and negative infinity. In this way, a range can have either one or both limits open. Second, a *ValueList* now replaces the *List-Based* class in previous version to represent a list of values of the same type. Third, the *Value* class, as already stated, has been modelled which can be classified into four main subclasses mapping to string, double, integer and float-based values. Actually, there are restrictions indicating that the value property that can be associated to each *Value* sub-class should map to the appropriate XSD type.

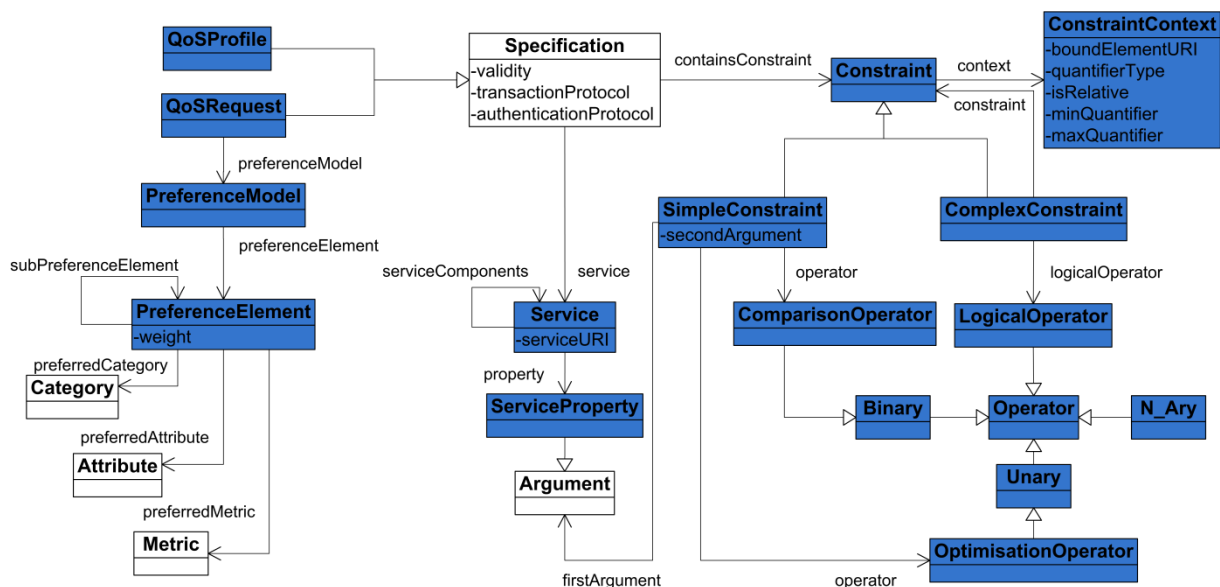


Figure 11 - The OWL-Q specification facet (with concepts coloured in blue)

Specification Facet. This facet is depicted in Figure 11. There have been particular updates with respect to the previous version. First, an actual renaming of some classes, like *QoSSelection* being renamed by *PreferenceModel*. The preference model is a tree-based structure which indicates the preferences as nodes (named as *PreferenceElements*) mapping to weights that the user has on certain quality terms. The main feature here is that the weights are relevant in the context of the same parent indicating the respective relative significance that each

term has with respect to a common reference point. For instance, by assuming that the current parent node maps to the quality category of *Performance*, the child nodes of *response time* and *throughput* can take the weights of 0.6 and 0.4, respectively. These weights indicate their relative importance with respect to their category. This hierarchical representation suits well the Analytical Hierarchy Process [113] and can be used for the ranking of services after being matched against a service request or in service concretisation problems to enable the proper formulation of the optimisation formula. Any specification is now connected to a particular Service which can comprise more simple services and which has particular service properties. Each service property is a kind of *Argument*, as already stated. Any service is also related to a specific endpoint (*serviceURI*) from which additional information can be fetched about this service (e.g., a specific mechanism is employed to obtain the interface specification of a SOAP-based service). In this way, we connect the non-functional specification of a service with the functional one without being tightly coupled with a particular functional service description language.

Any specification is associated to a *Constraint*. A constraint can be simple or complex. Complex constraints are associated to a logical combination of simpler constraints by applying logical operators like AND and OR. On the other hand, simple constraints are associated to a comparison operator (like GREATER_THAN), a threshold and an argument indicating the quality term on which the threshold (i.e., low or upper bound) should hold. Such constraint type is also associated to a *ConstraintContext* which indicates particular details concerning: (a) the URI of the element/object/component that is measured again catering for language independence but also for covering components in different layers; (b) similar information to CAMEL corresponding to the way the instance measurement level should be addressed (see respective description in section 2.2.1).

To conclude, we would like to stress that OWL-Q has been substantially extended to become more compact and easy to exploit, it supports different modes of modelling, it covers additional validation and knowledge derivation cases and more importantly covers all the layers in the extended cloud computing stack.

2.3.2.2 SLA Extension

Inspired by the survey in [16] which indicates the inability of current SLA languages to cover the information needed to support many of the activities of the service lifecycle as well as by considering the current needs of the project, OWL-Q specification facet was extended [24] in order to include a sub-facet, named as Q-SLA, focusing on the specification of SLAs. A snapshot of this facet is depicted in Figure 12. The design of this sub-facet relied on the main evaluation criteria of the SLA languages in [16]. These criteria map to the information that needs to be covered in each activity of the service lifecycle. As advocated in and shown in the following table, now Q-SLA outperforms all the current state-of-the-art SLA languages.

Life-cycle Activity	Criteria	WSLA [WSLA]	WS-A [WS-A]	WSOL [WSOL]	RBSLA [RBSLA]	LUA [LUA]	SLALOM [SLLLOM]	Q-SLA [Q-SLA]
Description	Formalism	Informal	Informal	Informal	RuleML Ontologies	Ontology	UML	Ontology
	Coverage	[p,y]	[y,p]	[p,p]	[p,y]	[y,y]	[p,y]	[p,y]
	Reusability	yes	yes	yes	yes	yes	yes	yes
	Composability	no	fair	no	no	no	no	fair
Matchmaking	Metric Definition	yes	no	no	yes	no	yes	yes
	Alternatives	impl	impl	impl	impl	no	no	yes
	Soft Constraints	no	yes	no	no	no	no	yes

	Matchmaking Metric	no	no	no	no	no	no	yes
Negotiation	Meta-negotiation	poor	fair	poor	poor	poor	poor	good
	Negotiability	no	part	no	no	no	no	yes
Monitoring	Metric Provider	yes	no	yes	no	yes	no	yes
	Metric Schedule	yes	no	no	yes	yes	no	yes
Assessment	Condition Evaluator	yes	no	yes	no	yes	no	yes
	Qualifying Condition	impl	yes	no	no	yes	no	yes
	Obliged	yes	yes	yes	yes	yes	yes	yes
	Assessment Schedule	yes	no	no	no	yes	no	yes
	Validity Period	yes	no	no	yes	yes	no	yes
	Recovery Actions	yes	no	yes	yes	no	no	no
Settlement	Penalties	no	SLO	SL	SL	SLO	SLO	SLO
	Rewards	no	SLO	no	SL	SLO	no	SLO
	Settlement Actions	yes	no	no	yes	no	no	yes
Archive	Validity Period	yes	yes	no	no	yes	yes	yes

Table 1 - The evaluation of Q-SLA against more representative state-of-the-art SLA languages

The root level concept of this facet is named *SLA*. This concept is a sub-concept of *Specification* highlighting that an SLA is a kind of non-functional specification. An SLA template in turn is a specific kind of an SLA. An SLA is associated to a validity period as well as a specific transaction and authentication protocol. It includes one more service levels (*SLs*) which represent the different performance levels that can be exhibited by the corresponding service and are relevant for this SLA. As such, *SLs* can be considered as a special kind of a composite constraint. A special kind of *SL* called *MaintenanceSL* was also modelled to cover the performance level of a service during maintenance periods. The SLA permits the transitions from one *SL* to another. The transition to a maintenance *SL* can occur in different ways: (a) on demand; (b) in certain periods; (c) both previous ways applying. On the other hand, the transition from one normal *SL* to another can occur when either: (a) the respective signatory entity requests this and is entitled to do so; (b) when certain conditions occur within a specific time period (such as a violation of a number of Service Level Objectives (*SLOs*)). Through this transitioning, we enable the specification of more flexible *SLAs*, which do not have to be re-negotiated when certain critical circumstances occur. For instance, when the service client needs to cover an additional load (e.g., due to an increase in the number of its customers), then he/she can indicate his/her intention to upgrade the *SL* offered by the respective service. As another example, if the service provider cannot anymore guarantee the delivery of a certain *SL*, he/she can indicate that the current *SL* is downgraded to a lower one. This downgrading will of course have an effect on the pricing of the service that will be reduced.

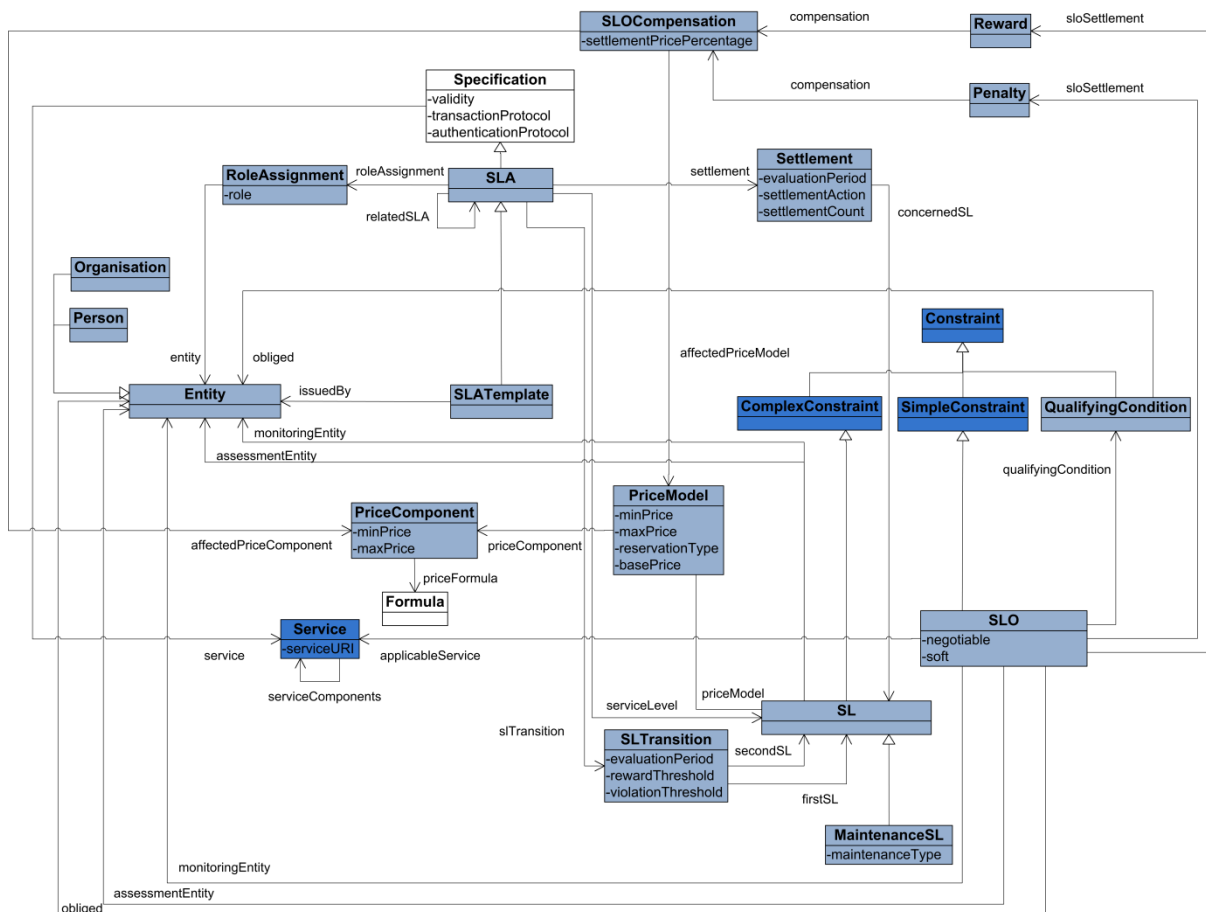


Figure 12 - Q-SLA sub-facet (with concepts coloured in light blue)

SLA usually includes a conjunction or a logical combination in general of SLOs. Each SLO is a kind of simple constraint for which additional information is provided. This information spans the following: (a) a qualifying condition indicating when this SLO should hold and is valid. Such a condition may include limitations on the side of the requester, which can include bounds on the frequency via which requests are issued; (b) a settlement should apply if the SLO is violated or surpassed. A SLO violation should map to a penalty, which is expressed as a discount over the normal service price while an SLO overpass to a reward expressed as a small percentage of increase in the basic service price. We believe that both types of settlement should be considered in SLAs in order to encourage service providers to increase the levels of service that they provide. This will lead to additional competition between the providers, resulting in better products and better prices for them; (c) the obliged party/entity which should be responsible for the satisfaction of the SLO; (d) the services or service parts for which the SLO should hold; (e) the parties responsible for the monitoring and assessment of the SLO; (f) negotiation/discovery-oriented information indicating whether the SLO is negotiable and/or soft. Negotiable SLOs are usually included in SLA templates indicating those quality terms for which the value can be negotiated. Soft SLOs are SLOs that are not obligatory in the sense that the service client can tolerate a violation of them.

Speaking about service clients, an SLA includes a set of *RoleAssignments* which indicate the roles that entities can play in this SLA which map to certain responsibilities. In this way, we can specify that Org₁ can play the role of service provider in SLA₁ and the role of service client/requester in SLA₂. As such, we allow the same organisation to play a different role in different SLAs. The types of entities that have been modelled include *Organisations* and *Persons*. The main roles have also been modelled mapping to PROVIDER, REQUESTER and THIRD_PARTY (mapping to an entity that might be obliged to perform particular tasks in the context of an SLA such as SLO monitoring and/or assessment).

A SL is also associated to a specific price model from which the price of the service can be derived. A price model maps to a certain monetary unit (e.g., EUROS) as well as to low and upper price bounds which indicate the highest and lowest limits that the price of the service can reach, irrespectively of the way this price is calculated. A price model is associated to one or more price components which explicate the way the price can be computed for a specific aspect. In this way, the total price of the service would be the sum of the prices derived from these aspect-specific components. Each price component is associated again to lower and upper price limits as well as to a price computation formula which is a kind of metric formula. In this sense, price can be considered as a metric that can be computed from other arguments, including particular features of a service. As a more concrete example, imagine the price for an IaaS service. This can include formulas over the computational, network and storage resources exploited that each can map to different price components.

When an SLO is violated or surpassed, this violation has an effect on the price model of the respective SL and especially on the corresponding price component that is affected. As already indicated, for the time being, a penalty or reward is expressed as a percentage over the price derived from a price component. In the future, we will consider other types or ways to compensate for an SLO violation or surpassing.

To cater for the modelling of critical situations which need to map to the performance of corresponding actions, like re-negotiation or SLA cancelling, a SL is associated to one or more Settlements which capture these critical situations and the actions that need to be performed. Currently, we consider that settlements can be expressed for lower-level SLs for which either a specific amount of SLOs has occurred in total or for a specific time period.

Finally, it has been decided that Q-SLA should follow a lightweight approach towards the capturing of SLA hierarchies that can well occur in the case of BPaaS. For example, a SLA for a BPaaS can involve the BPaaS broker and the BPaaS client while the SLA for the IaaS/PaaS/SaaS services that provide support to the BPaaS can involve either the BPaaS broker or client and the providers of these lower-level services. This lightweight approach maps to the modelling of the *relatedSLA* object property which relates one SLA with another one. This approach has been followed due to the increased complexity needed to fully specify such hierarchical SLAs and especially the various dependencies between the components and respective quality terms in the different levels. However, this decision can be modified in the near future, possibly in case a specific requirement is raised in the context of this project.

2.4 Future Work

The aforementioned sections have reported the current developments with respect to the modelling aspect for BPaaS allocation, execution, monitoring and adaptation. However, the project considers particular research directions that are going to be followed in the next period of the project. These research directions are now shortly analysed in separate sub-sections, which are grouped according to the sole language that they concern (CAMEL).

2.4.1 CAMEL Adaptation

Until now CAMEL covers only an application's scaling behaviour in terms of scalability rules that can be triggered to perform the respective scaling actions. However, in the context of this project, this adaptation type is restrictive and we need to expand on it to cover higher levels, i.e., the PaaS, SaaS and WFaaS levels in the context of CAMEL. For these levels, different types of adaptation actions apply and different conditions might need to be captured. In addition, as advocated by corresponding research results from other projects (S-Cube) and respective research work [106, 114], adaptation should be performed in a cross-layer and not a layer-specific and individual manner. This gives rise to the ability to interconnect conditions occurring at different layers as well as the actions that have to be performed to alleviate them. CAMEL can guarantee condition interconnection but this is not the case for adaptation actions. In our view, adaptation actions might have to follow a more complicated workflow rather than a simple one comprising just a sequence of two actions. In this sense, CAMEL needs to be extended such that it can specify in a direct or indirect manner such workflows. It should also allow specifying actions that need to be

Copyright © 2016 UULM and other members of the CloudSocket Consortium
www.cloudsocket.eu

performed at different layers. In case adaptation-based exceptions occur, it also need to supply alternative or compensation logic. All aforementioned extension directions will be followed for CAMEL by extending the current content of the SRL meta-model/language.

2.4.2 CAMEL Semantic Annotations

While not being a top priority for now, semantic annotations will greatly assist in two main BPaaS lifecycle activities: (a) adaptation and (b) evaluation. Adaptation in sense of substitution or re-composition requires to know the functionality of a BPaaS workflow task to be replaced. To enable an accurate substitution, this functionality should be semantically annotated (with annotations spanning the I/O parameters and possibly the functionality itself). As a SaaS usually points to an entry in the service registry, we believe that CAMEL does not need to be extended to cover this type of annotations. On the contrary, we believe that the service registry should be semantically annotated to support semantic service discovery by allowing the production of those semantic specifications that are needed by a semantic service discovery algorithm (see section 3.2.1).

Concerning BPaaS evaluation, measurement information needs to be semantically annotated so as to be semantically lifted to cater for the main analysis goals of the BPaaS Evaluation environment. Such an annotation could be done in two different ways: (a) semantically annotating metrics when specified in CAMEL or (b) semantically annotating the measurements. The first way enables to indirectly connect the annotation with the respective measurement produced via the metric identifier while the second way maps to a direct connection. The first way does not require modifying the measurement logic while the second way does as the respective sensor should be configured with the URI of the semantic metric specification.

Based on the above, CAMEL is going to be extended mainly as far as metric specification is concerned. Any other kind of annotation burdens mainly the contents of the different types of registries that are available (especially mapping to the description of PaaS, SaaS and IaaS services), provided that information at the higher business level is already semantically annotated.

3 ALLOCATION ENVIRONMENT BLUEPRINT

The main goal of the Allocation Environment is to produce a BPaaS bundle out of a BPaaS design package that is given as input to it. Such a BPaaS bundle should contain the concrete deployment plan which will be used in order to guide the deployment of the BPaaS by the Execution Environment. Such a deployment plan indicates the allocation decisions that have been taken. Such decisions concern the usual three layers in the cloud: (a) SaaS services are mapped to service tasks in the abstract workflow of the BPaaS design package; (b) PaaS and IaaS services map to the deployment of VMs (IaaS) or environments (PaaS) which will host the internal service components of the BPaaS. In this sense, these decisions concretise the abstract BPaaS workflow as well as cater for providing an underlying infrastructure support to guarantee that enough resources are engaged for the deployment and execution of the concrete BPaaS workflow.

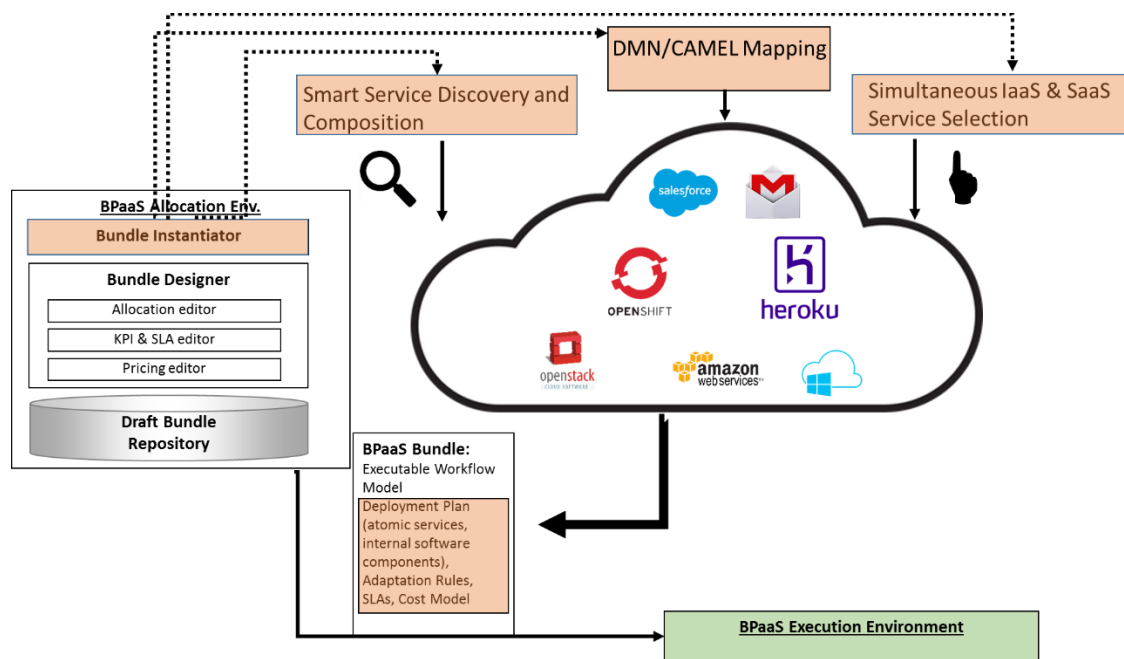


Figure 13 - Allocation Environment Blueprint

Figure 13 depicts the allocation blueprints, which will support the Allocation Environment in the required decisions. We believe that a step-wise approach should be followed which involves four main steps: (a) the discovery of those (SaaS) services that realise the functionality of the tasks of the abstract workflow in the BPaaS design package. In case that no single service can realise the functionality of a task, then service composition is executed to find suitable service combinations that do realise it; (b) the discovery of PaaS or IaaS services that satisfy the deployment requirements for internal service components of the BPaaS workflow; (c) the selection of one candidate SaaS service for each task in the abstract workflow; (d) the selection of one PaaS or IaaS from the candidate ones for each VM/environment in the deployment plan. We need to indicate here that the first two and the last two steps can be executed in parallel where the execution of the first two should precede the execution of the last two. Moreover, as the selection of services should reflect the satisfaction of broker non-functional requirements, we actually advocate that one combined algorithm should be in place in order to realise the last two steps. This is because the choices at the IaaS and PaaS level influence the QoS of the internal service components at the SaaS level and thus the overall high-level requirements that are posed at that level also influencing the selection of external SaaS services. In this way, a combined algorithm can guarantee the optimality of the solution derived while the execution of individual service selection algorithms at different levels will lead to a non-optimal solution.

In order to facilitate the generation of the BPaaS Bundle, the DMN Mapping describes the idea of semi-automatically generating parts of the CAMEL description for the actual service deployment. This will reduce the required technical knowledge about the cloud provider specific details and reduces the complexity for creating of the technical CAMEL description.

In the following, we first provide a short state-of-the-art analysis concerning approaches in cloud service discovery and composition. Then, we present the two research approaches that realise the four main steps in the BPaaS allocation approach. The description of the DMN Mapping Blueprint follows before we conclude this section by drawing directions for further research in the short and long term. Short term directions will be followed and reported in the next version of this deliverable while long term directions might be followed but is not expected to solve completely the main research problem to be addressed.

3.1 State-of-the-Art

A comprehensive state-of-the-art analysis for cloud service discovery and composition has already been reported in D3.1. In this section, we just shortly supply the main outcomes of this analysis.

Concerning cloud service discovery, we have indicated that there is a variety of approaches, which focus on different types of cloud services and on different aspects (functional and non-functional). For any kind of cloud service, approaches that employ semantics can reach higher levels of accuracy. The most promising approaches concerning functional SaaS service discovery [25] employ a combination of information retrieval and semantic web techniques. On the other hand, concerning non-functional SaaS service discovery the most prominent approaches come from the mixed category [26] in which QoS-based SaaS specifications are first aligned based on their QoS terms and then constraint solving techniques are exploited in order to perform SaaS matching. IaaS discovery seems to map to an easier problem and requires just the use of constraint solving techniques in order to perform the IaaS matching. In fact, in our view, IaaS matching looks similar to non-functional SaaS matching. This is because the main features of an IaaS offering can be seen as constraints on particular properties while the respective IaaS requests follow the same pattern. As QoS/non-functional capabilities and requirements are also expressed as constraints over QoS terms, in principle the same approach as in SaaS matching can be used in order to perform the IaaS matching. However, more sophisticated approaches can also be employed which map to the semantic-aware matching of feature models. Such approaches consider feature models, which can be regarded as more structured constraint models comprising constraints operating over the properties of features or across features. Moreover, feature models map to feature hierarchies. As such, they seem to match the different parts from which an IaaS offering can comprise.

Functional service composition has been applied mainly in the context of SaaS services. This is quite natural if we consider that, there is no meaning in composing IaaS services as usually abstract deployment plans are employed which comprise VM nodes mapping to a set of requirements that are then used to discover the most suitable IaaS services. In this way, the concretisation of abstract deployment plans is similar to the concretisation of abstract service plans where the selection of the best candidate IaaS service is driven by global non-functional requirements. Thus, by focusing on functional SaaS service composition, the proposed approaches can be distinguished into graph-based [27], model-based [28] and AI-based [29]. Model- and AI-based approaches are more automated but usually also slower than graph-based approaches. In addition, graph-based approaches seem to be able to cater for the variability in the workflow production by allowing the use of different types of workflow control flow elements apart from sequential ones. On the other hand, especially AI-based approaches can also apply semantics in order to guarantee a more accurate service composition result.

Service selection as already indicated maps to the concretisation of a plan, whether this plan is an abstract workflow or a deployment plan. Concerning SaaS selection or concretisation, the approaches can be split into the following categories: (a) semantic [30]; (b) heuristic-based [31]; (c) multi-tenant-based [32]; (d) variability and multi-criteria

decision making (MCDM) [33]; (e) aspect-based (e.g., focusing on network issues and availability of multiple SaaS instances) [34]. In the context of web service selection, the second and fourth categories have been mainly exploited. Heuristic-based approaches enable the production of sub-optimal solutions in a faster way while MCDM approaches are able to produce optimal solutions but suffer from performance problems. As advocated in [35], many approaches, irrespectively of their category, suffer from particular disadvantages concerning: (1) following a pessimistic or a optimistic approach covering the worst case or the average one and not all possible ones; (2) they consider that SaaS/web service offerings comprise single values for each QoS term while it is more proper, especially in dynamic environments, to model offerings comprising a range of values per term; (3) user requirements can be over-constrained leading to no solution - such a situation should be avoided by enabling to violation of least possible amount of user constraints in order to still propose a solution to the user; (4) usually there are dependencies between the QoS terms (metrics or attributes) that are not captured. Based on these disadvantages, a prominent approach [35] was proposed able to solve all of them.

In the case of IaaS selection, most of the approaches have focused on the so-called placement problem that regards the placement of VMs in a specific cloud. The approaches that do solve the exact problem that we are facing actually employ similar techniques like those used for SaaS concretisation, which was also evident from the above analysis. It is worth to mention that fuzzy-based [36] and stochastic learning techniques [37] have also been employed. The former attempts to map user-provided optimisation rules into an optimisation formula, catering for the fact that some users are not experts in deriving such formulas by themselves. The latter relies on the fact that insufficient information is considered for the IaaS selection (e.g., high-level performance goals and how they are mapped to low-level ones are not taken into account), which maps to supplying results which satisfy low-level requirements but are not suitable with respect to the expected application performance. This can be because either the IaaS provider may not conform to its promises or the type of IaaS service selected may not be actually suitable for hosting the respective application component. As such, the incremental learning approach learns from the previous execution history in order to improve the solutions proposed to the user.

3.2 Smart Service Discovery & Composition

In order to deal with the first two steps as indicated in the introductory part of this section, smart service discovery and composition algorithms have been developed covering both SaaS and IaaS services. In the sequel, we shortly analyse the respective research approach followed for each service lifecycle process by also providing a respective reference from which additional information and details can be found.

3.2.1 Smart Service Discovery

3.2.1.1 *Non-Functional Service Discovery*

We have developed various smart service matchmaking algorithms, which focus on the coverage of the non-functional aspect. These algorithms span the first (in the context of CloudSocket) [6] and third category of approaches (previous work) [26] in non-functional SaaS matchmaking. To remind the reader, the first category employs ontologies and uses subsumption reasoning in order to infer the matchmaking but is able to address only unary-constrained non-functional service specifications. All the developed algorithms rely on OWL-Q (see section 2.3). So they do account for the semantics in the description of the quality terms thus catering for higher accuracy levels as they are accompanied by a respective non-functional service specification alignment algorithm [38].

All the developed algorithms attempt to smartly organise the service advertisement space in order to enable a faster service matchmaking. They also rely on the matchmaking metric of conformance indicating that a non-functional request matches the non-functional offer when each solution of the offer is included in the solution space of the request. The algorithms of the third category transform the conformance matchmaking into constraint satisfaction while the algorithms in the first category transform it into ontology-based subsumption. In the first case, by having

each non-functional service specification transformed into a constraint model, conformance is expressed as follows: $\text{match}(P^D, P^O) \leftrightarrow \text{sat}(P^O \wedge \neg P^D) = \text{false}$. This means that the constraint model P^D of the request matches the constraint model P^O of the offer if and only if the constraint model constructed from the constraint model of the offer and the negation of the constraint model of the request is not satisfiable/feasible, i.e., it does not have any solution. In the second case, the conformance is expressed as follows: $\text{match}(S^D, S^O) \leftrightarrow S^D \supseteq S^O$, where S^D represents the ontology specification of the request and S^O the ontology specification of the offer.

In the following, we first shortly analyse the architecture of the research prototype developed and then we describe each algorithm classified based on the two concerned categories. In the end, there is a discussion of which algorithm to select for different circumstances.

3.2.1.1.1 Prototype Architecture

The architecture of the consolidated system is shown in Figure 14. The actual matchmaking and registration processes are analysed in the following sub-sections as they map to details that are specific for each category of algorithms. The analysis now concentrates on the functionality exhibited by each component of the architecture.

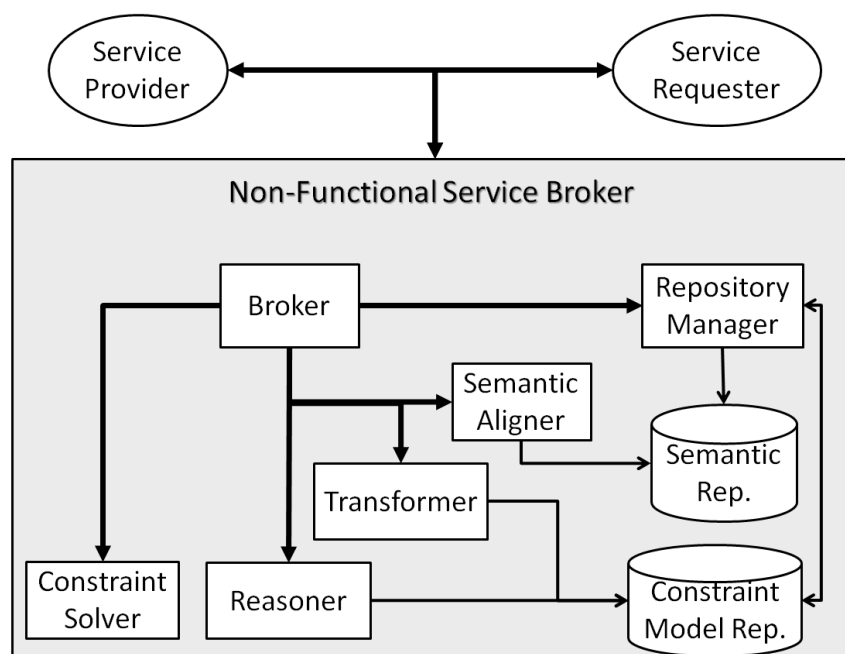


Figure 14 - The architecture of the non-functional service matchmaking prototype

The *Broker* is the entry point in the non-functional service discovery system, which is responsible for the orchestration of respective capabilities of other components in order to realise the service matchmaking and registration processes. In case invalid requests are issued, this component raises an error to the user/client.

The *Constraint Solver* is responsible for checking the consistency/feasibility of constraint models and employs different constraint solving techniques depending on the type of constraints involved [39]. Linear constraints are handled by mixed-integer programming techniques while non-linear ones with constraint programming techniques. Please note that constraint-based matchmaking is translated into constraint model consistency checking so this component is also exploited for matchmaking constraint-based non-functional service specifications.

The *Semantic Aligner* is responsible for aligning the non-functional service specifications issued by mapping their terms to the terms of a basic *Term Repository*. After the mappings are derived, the alignment involves a

transformation of the specification to include only basic quality terms which is performed by the *Transformer*. The *Transformer* is also responsible for transforming an ontology-based specification to a constraint model.

The *Reasoner* is responsible for checking the syntactic and the semantic validity of the specifications issued. It is also responsible for performing subsumption reasoning over an ontology (which can include two or more specifications). Both normal and incremental subsumption reasoning is supported.

The *Repository Manager* is finally responsible for the storage and retrieval of various artefacts that are stored in different repositories. These artefacts include semantic quality terms and semantic specifications stored in a *Semantic Repository* as well as constraint models which are stored in a *Constraint Model Repository*.

3.2.1.1.2 Mixed Category Algorithms

Four (existing) algorithms are exploited in this category. In the following, we first explain the main procedure followed for each algorithm in the context of advertisement matching and registration and then we analyse the main logic for each algorithm. More technical details about the exact logic, the complexity analysis and the evaluation of the algorithms can be found in [26].

The usual procedure for matchmaking is that the ontology-based user specification is first checked for validity. This means that an ontology reasoner is employed in order to check the syntactic and semantic consistency of the specification. If the specification is not valid, an error is relayed back to the user. Otherwise, the user specification is aligned [38] based on its quality terms against a repository of basic terms that have already been encountered. This avoids having to perform pair-wise comparisons of a request with all the offers currently stored in order to align it and reduces the possible term-to-term mappings that have to be considered in the alignment. Then, the user specification is transformed into a constraint model which is checked for consistency. If it is not consistent, an error is sent back to the user. Otherwise, the constraint model produced is matched against the constraint models of the offers stored in the service repository.

Concerning offer registration, the same steps as in matchmaking are performed in order to align and validate the ontology-based offer specification. Then, this specification is just registered based on the logic of the respective algorithm and especially the way it organises the service advertisement space.

3.2.1.1.2.1 Naive algorithm

This algorithm does not conduct any specialised organisation of the service advertisement space. In this sense, requests are matched against all offers registered via pair-wise comparisons. In this sense, this algorithm has the worst time in service matchmaking and the best for service registration as it does not have to perform anything else apart from just storing the constraint model of the offer in the service repository. Thus, it actually represents the main (performance) extremes in service matchmaking (worst) and registration (best).

3.2.1.1.2.2 Unary algorithm

The main logic of the unary algorithm is that it tries to organise the service advertisement space by using smart structures which consider just the thresholds of each non-functional service specification. A set of same structure instances is actually employed, each mapping to a different QoS term. As such, this means that this algorithm can only operate on unary-constrained non-functional specifications.

In this algorithm, the matchmaking is performed by considering each constraint of the user request. In particular, each user constraint is checked with respect to those offers that satisfy it. This results in a sub-set of offers, which have to be concatenated with the respective sub-set produced from the processing of the previous constraint in order. In this way, through set concatenation, we can reach a point where either no offer is able to satisfy all the constraints of the request that have been processed (so far) or all constraints have been processed and a set of offers matching the request have been found. As the derivation of a sub-set based on a user request is fast and this holds for the concatenation of sets, this algorithm is the fastest in service matchmaking.

Concerning service registration, each constraint of the offer is processed at a time and leads to the update of the structure mapping to the quality term that this constraint involves. This again leads to a quite fast service registration process but is not as fast as in the case of the naive algorithm.

3.2.1.1.2.3 *Subsumes algorithm*

The main idea here is that the service advertisement space is organised into a subsumption hierarchy where at the root we have nodes which are not subsumed by any other node and at the rest of the levels we have nodes that are subsumed by their parent while they subsume their descendants. An example of such subsumption hierarchy can be seen in Figure 15, where offer O_1 subsumes offer O_2 which in turn subsumes offers O_3 and O_4 .

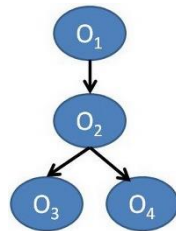


Figure 15 - An example subsumption hierarchy

Through such a subsumption hierarchy, service matchmaking can be fast due to the following observation: if a request subsumes a node, it also subsumes all the descendants of this node. In this way, we do not have to go down in the tree hierarchy in order to perform pair-wise comparison of the request with each descendant node.

Service matchmaking follows the above rationale. We compare the request first with each root node. In case of a match, we just collect the node's descendants and we include them in the matchmaking results (along with the root node). Otherwise, we need to go down the tree of the current root node because there is a possibility that a descendant mapping to a more strict constraint model matches the request. As such, in this case, matching is performed in a recursive manner.

Service registration follows a similar process as in matchmaking by starting with the root nodes. However, the difference now lies on the fact that we need to check all subsumption directions in a pair-wise comparison. This maps to covering the following cases: (a) the offer subsumes a root node but is not subsumed by it. This means that the offer becomes the parent of the root node and the processing continues; (b) the offer is equivalent to the node. In this case, it enters the node's equivalence set that maps to all offers that are represented by this node. The registration ends here; (c) the offer is only subsumed by the node. In this case, we have to recursively visit the descendants of this node; (d) the offer is not related to the node - the processing goes to the nodes siblings.

This algorithm might have good matchmaking performance if the subsumption hierarchy has more than 2 levels and certainly outperforms the naive algorithm in most of the cases. However, this algorithm has also the worst performance with respect to service registration due to also to the need to cover both directions of subsumption.

3.2.1.1.2.4 *SubsumedByAlgorithm*

This algorithm has similar logic with the previous one. The main differentiation lies on the fact that it relies on the opposite relation, *subsumedBy*, in order to organise the service advertisement space. The main rationale is that if the percentage of offers being matched is always low, then matchmaking will be faster than the *Subsumes* algorithm. This is achieved by the fact that if a request does not subsume the root node of a tree in a hierarchy, then it does not also subsume any of its descendants.

We do not shortly detail the service matchmaking and registration logic of this algorithm as it is equivalent to the previous one. We just need to indicate that this algorithm was indeed proven to be faster in service matchmaking with respect to the previous one when the percentage of matched offers was equal or lower to 0.3.

3.2.1.1.2.5 Discussion

Based also on the review in [26], the best algorithm is the *Unary* one as it is more scalable than the others in both service matchmaking and registration. However, as already stated, it only operates over unary-constrained non-functional service specifications. Thus, in case that n-ary constrained specifications are involved and registration time is not very important, then it is recommended that the subsumes-based algorithms are exploited where the *SubsumedBy* one seems to be more suitable based on the fact that it is more probable that a small percentage of offers is always matched against user requests.

3.2.1.1.3 Ontology-based Category of Algorithms

Almost the same procedure for service matchmaking and registration is followed by the category for which we have developed three main algorithms [116]. The only differentiation lies on the fact that the service specifications do not need to be translated into constraint models but they are matched or registered based on their original but aligned form. Constraint feasibility is guaranteed via ontology reasoning in this case. In the following, we shortly present the developed algorithms and then we provide a small discussion about which one to select in different circumstances.

3.2.1.1.3.1 Naive Algorithm

The main idea of this algorithm is that we use ontology subsumption as it is. This means that: (a) a new offer is just included in the existing specification of all offers registered - i.e., one ontology encompassing all offers; (b) a request is temporarily included in the former ontology and subsumption reasoning is just employed in order to discover those offers that it subsumes. In this sense, we expect that service registration will be the fastest possible while service matchmaking the worst possible with respect to the other algorithms proposed, especially as ontology subsumption does not scale well when the size of the ontology increases. As in the case of the naive algorithm in the mixed category of approaches, the same different extremes actually map to this algorithm, thus also justifying accordingly its name.

3.2.1.1.3.2 Incremental Algorithm

This algorithm tries to solve the main problem exhibited by the previous one. It considers the idea that incremental reasoning can be employed in order to decrease the matchmaking time. In this way, registration in this algorithm is almost equivalent to the previous one with the sole difference that now when every X offers are registered, incremental subsumption reasoning is performed. Matchmaking is performed as in the previous algorithm - we temporarily add the request to the offer ontology but we now perform incremental reasoning. The evaluation results have shown that this algorithm is worse in registration time than the previous one but is better in matchmaking time, reducing this time to one third in some cases.

3.2.1.1.3.3 Subsumes Algorithm

This algorithm has exactly the same rationale as in the case of the equally named algorithm in the mixed category of approaches. The only difference is with respect to the way conformance/subsumption checking is performed, where now pair-wise ontology subsumption is employed. Compared to the previous algorithms, this algorithm has a far better matchmaking time. Its registration performance is worse than the naive is but better than the incremental from a certain number of offers registered and above (450 in the experimental evaluation performed in [116]).

3.2.1.1.3.4 Discussion

Based on the above analysis, the best algorithm seems to be the subsumes one, if we especially consider that matchmaking time is more important than registration time. This algorithm is also more stable and scalable with respect to the previous ones.

3.2.1.1.4 Overall Discussion on Non-Functional Matchmaking Algorithms

While we have indicated the cases where one algorithm from those available in an approach category should be selected, we need to also provide some global recommendations spanning both approach categories for which

algorithms have been developed. In our opinion, the mixed-based approach seems to be faster and more scalable, especially in the case of the *Unary* algorithm. In this sense, if unary-constraints are only employed in specifications, we recommend the exploitation of this algorithm. In case that n-ary constrained specifications are involved, then different algorithms can be preferred. This depends on various factors, including the implementation technology. We expect that if state-of-the-art constraint solvers are used, then the *SubsumedBy* algorithm of the mixed category will be the best, as it will reach a better performance level in matchmaking and registration than its respective counterpart in the other category. However, in our evaluation, we have experienced some scalability and performance problems when employing free constraint solving components. Such problems lead to nominating the *Subsumes* algorithm in the ontology-based category as the best when exploiting the Pellet state-of-the-art open-source ontology reasoner [40]. Nevertheless, we expect that only unary-constrained specifications will be considered in the project and thus the use of the *Unary* algorithm will be the most appropriate and the one finally recommended.

3.2.1.2 Functional Service Matchmaking

Currently, we support functional service matchmaking by relying on two main algorithms. The first algorithm, called *Alive*, has been proposed in [Alive] while the other algorithm, called *Simple*, has been developed in the context of this project. Both algorithms employ information retrieval and semantic web techniques to perform the functional service matching and return as a result categories of matches based on the classification in [25]. Both algorithms require that services are functionally described via OWL-S but in the future they could adopt even other formalisms (by realising the respective processing or model transformation functionality needed).

Both algorithms attempt to organise the service advertisement space in a smart way in order to speed-up service matchmaking. The first algorithm employs smart graphs which enable in $O(1)$ time to retrieve the subsumption descendants of one node, where each node maps to a domain ontology concept. Both direct and indirect subsumption retrieval is supported. Via such graphs along with hash-based structures that map each I/O concept to the service that features it, matchmaking can be performed by retrieving for each output concept of the request the set of services, which produce output concepts that are subsumed by this concept. This set is concatenated with the respective set produced for the previous output request concept in order of processing. Thus, we reach a point where either the request is not matched by any service, as we end up with an empty concatenation result, or it is output-matched by a set of services when all output request concepts have been processed. The final set, if not empty, is then matched based on the user request input where the order of matching now is reversed. This means that we check if each input concept of the offer subsumes any input concept of the request. Now, input checking is performed on the fly as we expect that a quite small fraction of services is matched.

The second algorithm follows a similar matchmaking logic with the first one. The sole difference is with respect to the type of structures exploited. In this second algorithm, instead of the smart graph, the whole subsumption hierarchy of an ontology is mapped into a hash set which covers all direct and indirect subsumption relationships between pairs of domain ontology concepts. Compared to the graph structure, this second structure is static and thus cannot handle domain ontology updates. As such, the first algorithm is more robust to changes in domain ontologies.

Concerning offer registration, the procedure is quite similar for both algorithms. They first check if the offer is valid/consistent, a prior step also for service matchmaking. Then, they process each I/O concept of the offer in order to update the respective structures. The registration process takes usually longer time than the matchmaking one as ontologies have to be loaded, reasoned and the respective subsumption hierarchies must be incarnated into appropriate structure content. On the other hand, during the matchmaking process, only the specification ontology is loaded while then the normal matchmaking takes place by relying on the subsumption information already gathered.

Due to its robustness as well as on the fact that the first algorithm is faster, it is selected by default in the prototype and is highly recommended. Evaluation results for this algorithm can be found in [41]. We just need to note that this algorithm has been modified to correct some minor issues that prevented it from reaching higher accuracy levels. In this sense, the evaluation results concerning accuracy will be better than those in [41].

3.2.1.3 *laaS Matchmaking*

Currently, we follow the approach that laaS matchmaking is equivalent to non-functional service matchmaking. In this respect, the algorithms provided in section 3.2.1.1 also function for this type of matching. The only requirement is that laaS offerings are described in OWL-Q in order to be matched. In the future, we will consider whether the constraint-based matching is enough or more sophisticated approaches (like semantic feature model matching) can be employed.

3.2.2 Smart Functional Service Composition

From those planning algorithms that have been proposed, we have selected the AI-based ones due to their capability to also handle semantics. From this category, an algorithm developed by FORTH [42] has been chosen which is able to address service planning by also considering the frame, ramification and qualification problems in service specifications. The latter problems have been acknowledged to lead to composition accuracy issues. As such, the selected algorithm will certainly reach higher accuracy levels, provided that the service specifications are rich enough and are specified based on a specific XML-based language called WSSL (Web Service Specification Language) [43] that has also been proposed by FORTH. However, even if service specifications are not rich enough, the accuracy levels exhibited by the selected algorithm will be the normal ones as in other AI-based planners. Other features that make this planner more appealing are that: (a) it supports service validation/verification apart from service composition; (b) it is also able to produce plans, which are non-deterministic; (c) it is able to address information incompleteness due to partially observed service states.

3.3 Simultaneous laaS & SaaS Service Selection Algorithm

As it has been indicated in the introductory part of this section, SaaS and laaS selection should be performed in conjunction in order to take the best possible allocation decisions in all the levels involved that best satisfy the user/broker requirements posed. To this end, a constraint optimisation algorithm [115] has been developed able to perform this type of combined selection. This algorithm has been implemented based on the Choco constraint solver¹² and the Ibex constraint programming solver¹³ (for the internal handling of real variables). The main features of this algorithm are shortly explained below:

- It is able to consider global requirements on the overall performance of the BPaaS workflow as well as local requirements mapping to specific workflow tasks or nodes of the abstract deployment plan providing infrastructure support to this workflow.
- It is able to consider both high-level and low-level security requirements and capabilities. High-level security requirements and capabilities are represented by security controls while low-level security requirements and capabilities are represented by constraints on security-based (quality) terms (attributes and metrics). The high-level security requirements enable a coarse-grained filtering of the cloud service provider space while the low-level ones a more fine-grained filtering. Moreover, there is a connection between high and low-level security constraints in order to be able to infer how well a specific control is realised by a certain cloud service provider.

¹² choco-solver.org

¹³ www.ibex-lib.org

- It is able to handle non-linear constraints as well as constraints on sets. The former are appropriate for addressing quality terms like availability while the latter are suitable for handling quality terms with sets as their value type (e.g., coverage of areas for a map service).
- It is able to address over-constrained user requirements through the introduction of utility functions that enable the slight over-passing of the user-provided thresholds on quality terms. As such, it is able to produce solutions even for such (extreme) cases.
- It considers functions, which enable the propagation of QoS from lower- to higher-levels (e.g., a function that indicates that the service execution time depends on the underlying resource utilisation).
- It is able to consider cloud service offerings, which promise a range of values for each quality term and not just a single value thus catering for better capturing the variation of service levels in dynamic environments.
- It considers two different types of placement constraints (along with their opposite formulation): (a) two components should (or should not) be placed in the same VM; (b) two components should (or should not) be placed in the same cloud. The first type of constraint is suitable when two components have so frequent communication that is better to place them in the same VM. The second type of constraints is suitable when two components have frequent communication, which can be well supported based on the respective network characteristics of the same cloud in order also to reduce costs.
- Last but not least: It can take interesting allocation decisions when there is a different kind of variability in realising a particular functionality. In case that one functionality is internally supported via software/service components that have been already developed and externally by the existence of services that can be purchased, the algorithm can check whether it is more appropriate to use the internal component and host it in a specific well-suited cloud or to exploit an external service in order to realise the respective functionality.

The input to be given to this algorithm maps to the user global quality requirements as well as to local quality requirements posed over the tasks of an abstract BPaaS workflow plus VM attribute constraints over the abstract deployment plan supporting the BPaaS workflow. In addition, the algorithm needs to know all the possible alternatives in terms of: (a) services realising the functionality of a task; (b) VMs supporting the hosting of internal service/software components; (c) design choices with respect to some tasks based on the aforementioned allocation variation in the last bullet. The algorithm also needs to know the relative priority of one quality term over the others. In order to achieve that, the Analytical Hierarchy Process [45] is followed in order to derive a set of weights mapping to the desired global quality terms whose sum should equal to 1.

Based on all the above input, the algorithm creates a specific constraint problem and solves it via employing the aforementioned constraint solvers. While multiple objectives are given to this problem, through the assistance of the term-to-weight mapping, it can associate it to a single objective one which includes the following optimisation

formula: $maximize \left(\sum_{q=1}^Q w_q * uf_q (val_q) \right)$ (where q is the index of one quality term, Q is the number of all

quality terms, w_q is the weight given to this term, uf_q is the term's utility function and val_q is the global value that this term takes based on the selected solution). This formula indicates that the single problem goal is to maximise the weighted sum of the application of each quality-term-specific utility function to the global value that the respective quality term obtains according to the specific solution selected. This global value relies on the respective quality term values that the workflow tasks exhibit, the aggregation type of the quality term as well as the workflow structure. In this way, as cost is additive and does not depend on the workflow structure; the global cost value will be equal to the cost of all tasks. On the other hand, as availability is multiplicative, by considering that the workflow maps to one task execution sequence, the global availability value will be equal to the product of the availability values of all tasks involved in this sequence. In overall, the derivation of this global value is regulated by the following

equation: $val_q = f_q(val_i^q)$ (where f_q is a function mapping to the aggregation procedure of the q quality term and val_i^q is the value that this term takes for the i -th task in the workflow).

As already indicated, utility functions are formulated in such way that they map to enabling a slight deviation with respect to the range of values requested for the specific quality term. Depending on the term monotonicity, different functions are used which have a similar form. The formulation of these utility functions was inspired by the work in [35]. More details can be found in [115].

Three main decision variables are considered in this problem which also map to respective constraints. These regulate: (a) whether we select an internal service component or an external service to realise the functionality of a task; (b) whether a particular VM offering of a specific cloud provider from all candidate ones is selected in order to host an internal service component; (c) whether a particular service from the candidate ones is selected to realise the functionality of a BPaaS abstract workflow (service) task.

As already mentioned, we need to have a mapping from the QoS of a low level to the QoS of the higher level. Such a mapping is expressed via the following formula that connects a quality term of a workflow task to the QoS of the component used to realise it: $val_i^q = y_i * f_i^q(core_i, mem_i, store_i) + (1 - y_i) * \left(\sum_t z_{it} * val_{it}^q \right)$ (where y_i represents

the decision of whether we select the internal service component realising the functionality of the task, f_i^q is the function mapping the QoS of the internal service component to the QoS of the VM used to host it), $core_i, mem_i, store_i$ represent the amount of cores, memory and disk storage size of the VM used to host the service component, z_{it} represents the decision of whether we select a particular external service to realise the functionality of the task and val_{it}^q is the term value for this external service).

The amount of cores and the sizes of memory and disk storage are computed from the following formula (where x_{ijk} represents the decision of whether k VM offering from cloud j was selected to host the component realising the functionality of task i):

$$core_i = \sum_{jk} x_{ijk} * core_{jk}$$

$$mem_i = \sum_{jk} x_{ijk} * mem_{jk}$$

$$store_i = \sum_{jk} x_{ijk} * store_{jk}$$

This formula indicates that these characteristics are derived from the respective characteristics of the VM selected to host the internal service component.

Pair-wise placement constraints for the same VM are expressed via the following form: $x_{ijk} = x_{i'jk}$ where i and i' are the indices of the components to be hosted on the same VM. This expression indicates that the same allocation decision concerns both components. The pair-wise placement constraints for the same cloud are expressed as follows: $\sum_k x_{ijk} = \sum_k x_{i'jk}$. This expression indicates that a decision must be taken for two components which involves the selection of VMs in the same cloud.

More details about the whole formulation of the constraint problem can be found in [115] as we desire not to overwhelm the reader.

The algorithm has been also evaluated. The initial evaluation results show that a more optimal solution is reached with respect to solving two individual IaaS and SaaS selection problems. It also indicates that while the constraint problem size is bigger with respect to the corresponding problem for IaaS selection, the solving time of the combined selection algorithm is less than that of an IaaS selection algorithm. Finally, the evaluation indicates and validates experimental results from other research propositions which signify that the more is the size of the placement constraints, the less is the time needed to solve the combined service selection problem. This means that the deployment plan should be rich enough in order to enable a faster selection time by including many placement constraints, when such constraints can be applicable. However, please have in mind that the modeller should be careful not to pose many placement constraints as it also risks reaching the situation where the constraint problem produced is infeasible.

Finally, we would like to mention that in case the low-to-high level mappings cannot be expressed, the algorithm can still perform the different types of selection in an individual manner as this would make more sense. In this sense, it can be configured to perform just one type of selection and the user will be then responsible on the exact method to follow in order to coordinate the execution of the individual selections to be performed.

3.4 DMN to CAMEL mapping

Whereas the tool support in modelling and orchestration of cloud applications has been risen for the technical experts, including DevOps tools like Chef or cloud orchestration tools like Cloudiator, business experts still require technical assistance for consuming cloud services. Therefore we propose a novel approach to support business experts in consuming cloud services based on higher-level business values. As introduced in chapter 2.2, CAMEL allows the specification of cloud applications with respect to deployment, monitoring, scaling, cloud provider offerings, and security. Whereas a technical expert has the required knowledge in these areas, a business expert comes with a higher business view.

The Decision Model and Notation (DMN) [46] standard is a way to model decisions by the means of tree-based decision tables. The novel approach is to integrate DMN into the modelling environment ADOxx¹⁴, in order to semi-automatically generate CAMEL models that comply to the business requirements of the company and allow to bring dynamicity to the CAMEL models. Whereas several CAMEL models, along with the technical descriptions of the services, may comply with the same business process, they are basically independent and differ in the suitability to different business requirements. The description in the CAMEL is quite static and does not reflect decisions that has to be taken case-by-case for the business requirements of the companies. By having, a meta format of CAMEL that integrates DMN tables to dynamically reason about the actual used services and their properties, it will be able to ease the process of creating a BPaaS bundles that fit for a particular customer classes. This DMN-enabled CAMEL format will be used to create complete CAMEL models that can actually be deployed.

Figure 16 shows the possible integration points for DMN into the BPaaS life-cycle¹⁵. Point (a) is the mapping of a task to a service or service composition. The parameters for mapping would be functional and non-functional descriptions of the task and service. Point (b) is the mapping between a service and a deployment description. For this decision, the consideration of the SLAs of the customer and the properties of the cloud provider is necessary. Point (c) is the mapping of the DMN decision tables to the rules of the deployed workflow that decide on the service's behaviour on run-time.

¹⁴ <https://www.adoxx.org/live/web/cloudsocket-developer-space/space>

¹⁵ <https://www.cloudsocket.eu/common-understanding-wiki/-/wiki/Main/CloudSocket+Process+Terminology>

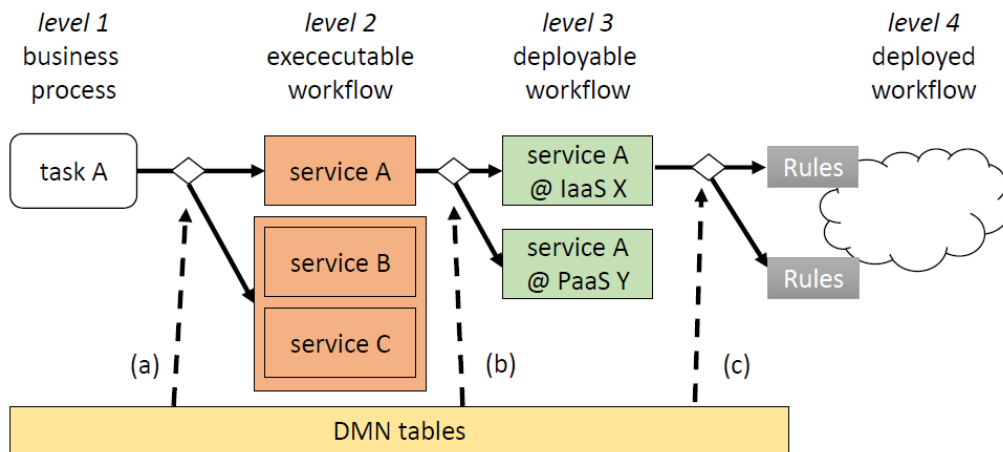


Figure 16 - Integration points for DMN into the BPaaS process.

The first approach for introducing DMN into the CAMEL creation focusses on (b). As business experts still requires technical assistance for consuming cloud services, our approach aims to create a way to semi-automatically handle the service selection and configuration based on high-level parameters.

Our approach enables the modelling of cloud applications by using non-technical business values, which will be mapped to a technical CAMEL model, by using *Business Knowledge Model (BKM)* in combination with *Decision Tables (DT)*.

We use a modelling environment to support the editing of DMN models, such as ADOxx¹⁶, in order to *program* the application deployment. The modelling environment will allow creating a meta-model on top of CAMEL including DMN references in a CAMEL model. These DMN references link to a set of DTs for each sub-set of CAMEL, e.g. the cloud provider offering for a given service. Each placeholder receives a set of business values as input for the referenced DTs. The referenced DT span a tree of DTs connected by their in- and outputs. The final output will be put in that place in the CAMEL model.

That means business experts define parameters that are important for a given task from a business view, which serve as input for the root DTs. A technical expert decides on how this influences the selection of services by defining the correlation to the service offerings with respect to specifying the output of the DTs that may serve as input for higher-level DTs. Currently, we assume this as a manual task and leave the semantic annotations for the semi-automatic generation of such DTs as future work.

In the top DT of such a tree, the output is the actual selection with respect to the CAMEL-part that was to be reasoned about.

3.4.1 DMN Mapping Scenario

In the following we present a sample scenario identifying an appropriate cloud provider by mapping business values to the concrete CAMEL specification for the cloud provider.

We have chosen the CloudSocket use case of the *Christmas Card Designer*. In our sample scenario, two companies use identical workflows and services but with different business needs with respect to *expected parallel customers* and *privacy level*.

¹⁶ <https://www.adoxx.org/live/web/cloudsocket-developer-space/bpmn-and-dmn-tool>

Company A wants to serve up to 100 customers with a high data privacy level, whereas company B wants to serve up to 1000 customers with a low level of data privacy.

A business expert defines the set of input variables, which might also be extracted from the knowledge space of a business requirement analysis that feeds the decision tables. In the case of our scenario, the selected output values will be mapped to the technical selection of the IaaS provider to host the application.

While the selection of an appropriate IaaS provider typically requires a technical expert considering characteristics like location, the cloud deployment model or virtual machine offerings, a business expert will consider higher level parameters as privacy level or number of expected customers. In order to select an appropriate IaaS provider model for the card designer service, we combine multiple DMN tables to transfer the business needs *privacy level* and *expected customers* into technical parameters, which map to the cloud provider model of CAMEL.

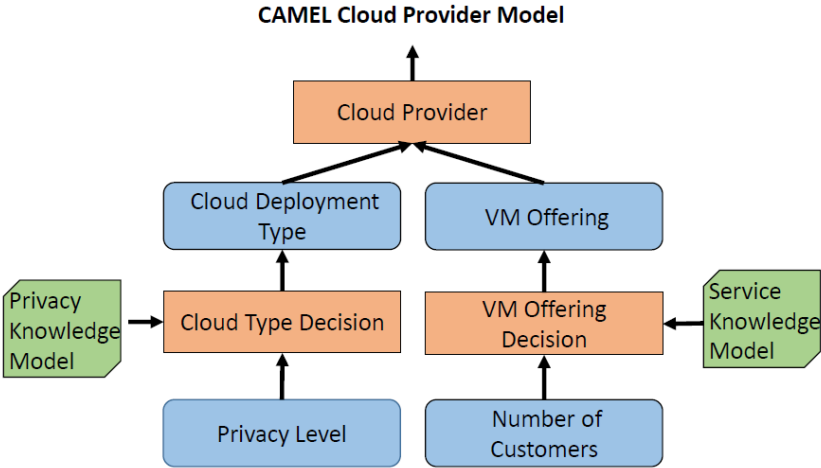


Figure 17 - DMN-to-CAMEL mapping.

An illustration of this approach is provided in Figure 17 comprising the business needs as input, an exemplary set of DT with the BKM. We only use the three decision tables, Cloud Type Decision, VM offering Decision and Cloud Provider Decision, in order to keep our approach comprehensible. Each decision table is defined by the respective BKM. Each BKM is reusable and needs to be defined beforehand by a technical expert.

A simplified DT in DMN notation¹⁷ is shown in Table 2.

Hit Policy	Input	Output	
C	Privacy Level	Cloud Type	Continent
1	Low	public	US
2	Low	public	Europe
3	Medium	private	US
4	Medium	public	Europe
5	High	private	Europe

Table 2 - Cloud Deployment Type DT

¹⁷ <http://www.omg.org/spec/DMN/1.1/>
 Copyright © 2016 UULM and other members of the CloudSocket Consortium
www.cloudsocket.eu

Based on the passed privacy level it maps to a list of tuples (cloud deployment type, continent). The resulting output is combined with the output of the VM Offering DT and passed to the Cloud Provider DT, which will map to a technical CAMEL model as depicted in Figure 17.

3.4.2 Identified Challenges

The presented approach covers a first set of necessities towards a dynamic description of cloud-based applications. However, there are some challenges that have to be examined in more detail. It is open, who creates the business knowledge model to feed the respective decision table. In addition, how the correlation is determined between a service (and its properties) and the available input parameters for a decision table.

In order to work in a modelling environment like the one mentioned above, a meta language upon CAMEL has to be defined, to handle the integration of DMN into CAMEL. A way is needed to transform from a non-runnable CAMEL to a fully self-contained CAMEL model.

A definition of semantic annotations and their mapping to lower level requirements is needed. For example, if the modeller decides on the type of input parameters, such as number of users or region, the system must know how this relates to a certain service or a property of a service.

It is not challenging for the region, since you can obtain the meta-information region from the service as it is, so this is a very simple mapping. However, for the number of users, it is more complicated to define the relation of this parameter to the properties of a service. Having said that, there is a need to meaningfully annotate the services and the properties that can be chosen as input parameter.

3.5 Future Research

While current research already performed in the context of BPaaS allocation can be considered quite fit to the purposes of the desired functionality, there are still some pending issues that will drive the short and long-term research to be pursued by the project partners. The issues are analysed in the following in separate sub-sections.

3.5.1 Combined Service Discovery

While service discovery algorithms focusing on a specific description aspect (functional or non-functional) have been developed and can be integrated into a combined service discovery research prototype, it is still pending to investigate the way these algorithms can be combined to realise a complete service matchmaking functionality. This is due to the fact that different combinations of aspect-specific algorithms can lead to different trade-offs between matchmaking and registration performance.

3.5.2 Overall Service Concretisation Method

While this section has unveiled the pieces that need to be integrated together to support the abstract to executable workflow mapping, there is a need to define a method that not only appropriately integrates them but also coordinates them accordingly. This can be done by checking what are the cases to be covered (e.g., M-1 or 1-1 task-to-service mapping) and what can be the required interaction from users and the input that can be provided by them. This method can also benefit from extensions of existing algorithms in order to cover initially not planned functionality as the one needed for service plan selection.

3.5.3 QoS Mapping Derivation

As indicated in section 3.3, the respective algorithm proposed requires that specific functions are derived indicating how QoS at the lower levels (IaaS) propagates to QoS at higher levels (SaaS). This mapping is valuable for investigating those IaaS services that are better suited for hosting internal services components covering some of

a BPaaS workflow's functionality. In fact, it has been increasingly reported that VMs with similar characteristics, offered in different clouds, tend to map to different component performance. To solve this problem, various type of techniques have been proposed like benchmarking and performance model learning [117]. Thus, it has to be investigated which technique is the most appropriate one to be exploited in the combined service selection algorithm and whether it needs to be extended accordingly.

3.5.4 PaaS Consideration in Discovery & Selection

Another direction to be pursued concerns how PaaS services can be exploited in service discovery and selection. By considering that PaaS can cover the functionality of particular components, like databases, and component hosting, it seems that PaaS is more appropriate to be considered as a potential replacement of IaaS services. As such, PaaS requirements can be inserted in abstract deployment plans to drive the infrastructure support to a BPaaS workflow. We believe that the handling of PaaS is more or less similar to the way IaaS services are handled. However, special care must be placed at the selection algorithm due to the additional level inserted which can further increase the complexity. Moreover, issues concerning how to derive PaaS performance and map it to the performance of components hosted by them are also relevant. Finally, the use of PaaS might also lead to employing more advanced matching of features models for IaaS and PaaS discovery.

3.5.5 Rich Service Specification

In the current situation at the service world, services are described just based on their respective interface. This leads to structural specifications which do not cater for high accuracy in service discovery and composition. To solve this problem, after such specifications are gathered, there is a need for semantic annotators that map the service I/O and offered functionality to concepts from a domain and task ontology. As such, we will investigate using or extending an automated service annotation approach so as to cover both types of annotations.

3.5.6 Formalism Transformation

Each algorithm exploited for BPaaS allocation relies on a certain service specification language. The functional service matchmaking algorithm relies on OWL-S, the non-functional ones on OWL-Q while the service planning algorithm on WSSL. As service specifications can be described in different languages from those expected, it might be decided to develop transformation functionality to enable transforming service specifications in the language expected by the respective algorithm.

3.5.7 Service Filtering

There already exist thousands of service specifications in the real world which can benefit the realisation of BPaaS. However, it is expected that not all available services suit the requirements of the broker as such services must be offered only from reliable and trustworthy cloud service providers. As such, service filtering algorithms are needed before service specifications enter the respective registries by considering suitable reliability and trust metrics which are derived based on: (a) the existence of formal contracts or SLAs guaranteeing a certain quality level for these services; (b) past performance or feedback from users or cooperating partners with the broker; (c) certain characteristics of the cloud provider (e.g., size, service variety, market share).

3.5.8 Semantic annotations for DMN Mapping

As mentioned above, a decision process via DMN, can also be applied to reason about the mapping between task and service, as well as using DMN as high-level language for the rule part of CAMEL, i.e., the SRL. For the first mapping, it is required to have semantic annotations for the functional and non-functional requirements for both, the task and the service, as described in D3.1. For the latter usage of DMN, we need to specify semantics for the SLAs and their correlation to adaptation actions, in order to automatically generate SRL rules and actions from the definition of a DMN decision table.

Semantic annotations will also support to ease the process of realizing the integration of DMN into CAMEL by the means of a modelling environment like ADOxx. This is, because of the semi-automatic generation of decision tables by the available input parameters, generated from the semantics of the service, one wants to reason about.

4 EXECUTION ENVIRONMENT BLUEPRINT

The Execution Environment is responsible to orchestrate, monitor and adapt the execution of the BPaaS bundles generated in the Allocation Environment, which have been published via the Marketplace. Hence, the main functional capability is to guarantee the execution and the suitable behaviour of the deployed BPaaS bundle. To this end, the environment is responsible to deploy, execute and re-configure the BPaaS Bundle to still satisfy the service level promised. Orchestration, monitoring and adaptation rely on the provided BPaaS Bundle specification. Figure 18 provides an overview of the Execution Environment components and the three main research directions. By focusing on the Orchestration, Monitoring and Adaptation across all cloud services levels, we are able to overcome current limitations and enable the execution of a holistic BPaaS lifecycle. In the following, the individual research assets for orchestration, monitoring and adaptation are presented. For the further components of the Execution Environment, i.e., Workflow Manager, Workflow Engine and Process Data Mediator, we have not foreseen any research challenges. Our focus lies on enabling the cross-cloud support for orchestration, monitoring and adaptation.

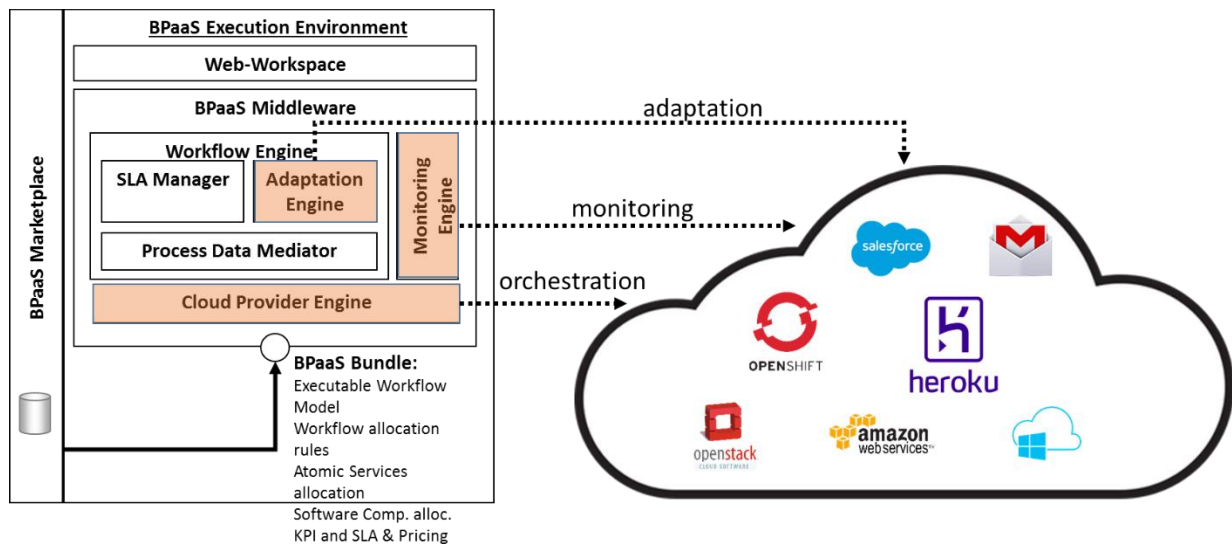


Figure 18 - Overall architecture of the BPaaS Execution Environment

4.1 Orchestration

The evolution of Cloud Computing has led to a state where the cloud paradigm has reached the mainstreams of software development and application operation. Nevertheless, many issues still have to be considered as unresolved. In particular vendor lock-in and limited auto-scaling capabilities are considered the most pressing and limiting aspects of cloud computing today [47]. Vendor lock-in avoids an easy migration from one cloud provider to another. It also avoids the parallel use of multiple cloud providers and establishes a technical barrier between operators and providers. In order to satisfy these demands, a powerful and reliable cloud orchestration and operation platform is needed. Indeed, there are multiple commercial and open-source tools available that promise to solve the aforementioned issues [48].

Whereas recent approaches focus on providing orchestration tools only for the IaaS level, the Cloud Provider Engine of CloudSocket aims to cover the orchestration across the IaaS and PaaS level [49]. The support of service orchestration over these cloud service levels enables the BPaaS paradigm via SaaS realise missing business

process functionalities, IaaS and PaaS provide the infrastructures and platforms to execute business process in the cloud.

This section describes the CloudSocket approach in developing a Cloud Provider Engine that orchestrates and abstracts cloud services not only for the IaaS but also for the PaaS level. First, an overview of the current state of the art of IaaS and PaaS abstraction layers and tools is provided. Second the Cloud Provider Engine, namely *Cloudiator* [50], is presented in its original state, as it was adopted from the PaaSage project. Based on this state, an evaluation of Cloudiator against other orchestration tools is performed, deriving the required extension to enable the BPaaS orchestration in CloudSocket. Finally, the required extension of Cloudiator to enable the complete BPaaS orchestration are presented.

4.1.1 State of the art

4.1.1.1 IaaS Abstraction Tools and Platforms

A first approach into IaaS abstraction is provided by IaaS libraries. Common representatives are Apache jclouds¹⁸, Apache Libcloud¹⁹ or fog²⁰. Apache jclouds is a Java based API abstracting more than 10 IaaS providers by also mapping their VM offerings to generic templates. In addition, it also supports a subset of these providers' storage APIs. Apache Libcloud is a Python based API abstracting the compute and storage APIs of more than 20 IaaS providers. Fog is based on Ruby and abstracts more than 15 IaaS providers. All of these libraries provide a single interface to users abstracting all the IaaS provider specific characteristics. By using such an abstraction layer, the provision and deployment of IaaS resources is facilitated which also eases the deployment of applications across different cloud providers, i.e., a multi-cloud deployment.

Whereas the abstraction layer APIs just focus on the resource abstraction and provisioning, cloud orchestration tools follow a more advanced approach. This approach combines the resource management with the full life-cycle management of the applications that are typically described in a DSL. Besides the application deployment, orchestration tools may also exhibit monitoring and adaptation features.

Apache Brooklyn²¹ is a framework for modelling, monitoring, and managing applications through autonomic blueprints. Apache Brooklyn provides the following capabilities: deploying to cloud and non-cloud targets; using monitoring tools to collect key health/performance metrics; responding to situations such as a failing node; adding or removing capacity to match demand²². A Brooklyn blueprint defines an application using a declarative YAML syntax. For example, a basic blueprint might comprise a single process, such as a web-application server running a WAR file or a SQL database and its associated DDL scripts. The types of supported entities are listed in the Brooklyn catalog²³. Currently, Brooklyn uses a YAML syntax which complies with the CAMP's one and exposes many of the CAMP REST API endpoints. On the other hand, an extension²⁴ has been developed to manage TOSCA blueprints, but this extension is not official yet.

Cloudify²⁵ by GigaSpaces Technologies is offered in a free open-source as well as a paid Pro edition. Cloudify uses a TOSCA-aligned modelling language for describing the topology of the application which is then deployed to allocated virtual machines in the cloud environment. As in TOSCA, Cloudify splits the blueprint in a type and a template definition. Types define abstract reusable entities that are to be referenced by templates. The types

¹⁸ <https://jclouds.apache.org/>

¹⁹ <https://libcloud.apache.org/>

²⁰ <http://fog.io/>

²¹ <https://brooklyn.apache.org/>

²² <http://brooklyn.apache.org/learnmore/theory.html>

²³ <http://brooklyn.apache.org/learnmore/catalog/index.html>

²⁴ <https://github.com/cloudsoft/brooklyn-tosca>

²⁵ <http://getcloudify.org/>

therefore define the structure of the template, by e.g. defining the properties that a template can have/must provide. The template then provides the concrete values. This mechanism is used for nodes as well as for relationships.

Apache Stratos²⁶ makes use of an abstract virtual machine description, named cartridge, with an application component type (named cartridge type) like an application runtime container (e.g. Tomcat). An application is described by a single cartridge or/and a set of cartridges (groups), combined with deployment and scaling policies. The cartridges, applications and other configurations are represented in an Apache Stratos specific JSON format. For the installation, it solely relies on the DevOps tool Puppet. The application itself is subsequently cloned from a Git repository. Stratos is installed as one central controller and in all virtual machines by having a virtual machine image prepared with the necessary software (Stratos and Puppet agents) installed.

The CloudML²⁷ approach [3] uses their DSL CloudML to describe the application. Based on the application description CloudMF supports the deployment and adaptation of applications across multiple cloud providers. Therefore they apply the models@runtime approach to align the actual application state with the desired CloudML state.

The main goal of MODAClouds²⁸ is to provide methods, a decision support system, an open source IDE and run-time environment for the high-level design, early prototyping and automatic deployment of applications on multi-Clouds with guaranteed QoS. The MODAClouds IDE allows the specification of cloud provider independent models of an application, together with the QoS to be fulfilled at runtime. This model-driven development enables applications to be deployed in multiple clouds –avoiding vendor lock-in–, to be monitored, and to be adapted in order to maintain the desired QoS. MODAClouds uses CloudML, a project that provides a domain-specific modelling language along with a run-time environment for the provisioning, deployment, and adaptation concerns of multi-cloud systems at design-time and their enactment at run-time. In the scope of the MODAClouds project, Tower 4Clouds was developed, a monitoring platform ready for the multi-clouds paradigm.

PaaSage is an open source integrated platform to support both design and deployment of Cloud applications, together with an accompanying methodology that allows model-based development, configuration, optimisation, and deployment of existing and new applications independently of the existing underlying Cloud infrastructures. The deployment approach of PaaSage overcomes the vendor lock-in by supporting multi-cloud deployments and abstracting the underlying cloud providers. In contrary to MODAClouds, PaaSage integrates the CloudML DSL into CAMEL, a new DSL for cloud requirements, deployment, adaptation and organisations. Further, PaaSage implements an extensible cloud orchestration engine that executes CAMEL deployment plans. CAMEL and the cloud orchestration engine are pursued and extended in the context of CloudSocket.

4.1.1.2 PaaS Abstraction Tools and Platforms

Whereas on the IaaS level, there is a relatively high common sense of what the providers offer, i.e. mainly computation, memory, storage and network in terms of virtual machines, the PaaS level encompasses a more abstract offering, i.e. environments or containers.

There are two categories of PaaS APIs [51]: (i) implementation API that caters for data storage, message queuing and similar capabilities and (ii) the deployment API that handles e.g. the container creation and configuration.

Hossny et al. [51] describe an approach of generating adapters for a generic PaaS-API based on semantic annotations of the specific PaaS API. This approach focuses on the implementation APIs. This is an enhancement of the approach of defining a single generic API that is manually implemented for each provider API, and updated when the specific provider API changes. Also a common API across different database providers was proposed in

²⁶ <http://stratos.apache.org/>

²⁷ <http://cloudml.org/>

²⁸ <http://www.modaclouds.eu>

[52] In the remaining part of the section, we will focus on abstractions of the implementation API, as this is the focus for PaaS abstraction in the first instance of CloudSocket.

A comprehensive taxonomy for comparing PaaS providers can be found in [53]. Kolb and Wirtz present also a standard profile for common capabilities of current PaaS offerings. They also propose a model for PaaS with three layers: infrastructure, platform and management. Moreover, based on that, PaaS can be categorized in IaaS-centric, generic and SaaS-centric PaaS, depending on the level of provided management and possible control of the platform. This will be used in the decision whether an application can be deployed on a certain provider, e.g. since the application demands for hard requirements towards the infrastructure, that cannot be guaranteed by all PaaS providers.

Sellami et al. [54] introduce (i) a unified description model allowing the PaaS provider independent representation of applications and (ii) a generic PaaS deployment API that is called COAPS API. It allows the specification of a manifest for the application and its environment, in a way that allows the deployment across multiple PaaS providers. Moreover, it provides a REST -ful API for the management (createApplication, destroyApplication, etc.) that internally calls the APIs of the actual chosen PaaS providers. Therefore, this approach provides a generic PaaS life-cycle for the PaaS deployment. Similar proposal exists also for the implementation APIs of PaaS platforms, such as for persistent storage in [53]

Walraven et al. propose a middleware for multi-PaaS environments called PaaS Hopper [55]. Here they introduce an abstraction layer that offers a uniform API to the application component to communicate to the underpinning cloud services via the middleware. The API is defined for structured storage, blob storage and asynchronous execution task; it therefore targets the implementation APIs.

Cloud4SOA²⁹ provides an open semantic interoperable framework for PaaS developers and providers, capitalizing on the Service Oriented Architecture (SOA), lightweight semantics and user-centric design and development principles. The Cloud4SOA system supports Cloud-based application developers with multi-platform matchmaking, management, monitoring and migration by semantically interconnecting heterogeneous PaaS offerings across different providers that share the same technology. All this is done using a user-centric web interface. The Cloud4SOA platform provides an API to manage the lifecycle of applications in PaaS clouds; this API is not implemented as a single library, but its architecture is composed of a local module with the main REST service and a set of Remote Adapters, which handle the complexity of the interaction with the cloud provider. These Remote Adapters were also in charge of the monitoring of the application, providing some basic metrics like *response time* and *availability*.

SeaClouds³⁰ provides a platform to enable seamless adaptive multi-cloud management of complex applications, by supporting distribution, monitoring and adaptation of application modules over multiple IaaS or PaaS clouds. The SeaClouds GUI allows the definition of an application and its QoS from a high-level and cloud-independent view, and offers a set of clouds where the application can be deployed. SeaClouds embraces TOSCA, and the final deployment blueprint is specified in this language. Apache Brooklyn is used as deployment engine, enriched with a TOSCA extension, to which SeaClouds collaborated in its implementation. Once the application is deployed, it is monitored with the Tower 4Clouds platform, developed in the MODA Clouds project. One of the results of SeaClouds is the PaaS Unified Library³¹, a library and REST service that provides simple operations for managing applications in PaaS providers.

²⁹ <http://www.cloudwatchhub.eu/cloud4soa-%E2%80%93-bringing-interoperability-portability-paas>

³⁰ <http://www.seaclouds-project.eu>

³¹ <https://github.com/SeaCloudsEU/unified-paas>

4.1.2 Cloud Provider Engine (Clouidiator)

The Cloud Provider Engine is responsible for the complete deployment and lifecycle management of all the required components of the BPaaS, including the management of their associated resources, e.g., VMs at the IaaS level or container/environments at the PaaS level. These capabilities are managed by different subcomponents of the Cloud Provider Engine to provide a modular, flexible and scalable architecture. To exhibit these capabilities, the Cloud Provider Engine is built upon existing functionalities offered through the interfaces exposed by the cloud providers.

The original version of Clouidiator [50] was developed during the PaaSage project with the focus on abstracting IaaS providers and enable the multi-cloud application deployment support.

4.1.2.1 Original Version

Clouidiator³² is a cross-cloud deployment tool that also supports adaptation and re-deployment. The deployment of Clouidiator features the capability to transform applications into application instances and store them in its internal application component registry. The deployment specification is described in CAMEL (cf. section 2.2).

A general overview of Clouidiator is depicted in Figure 19 where the green entities mainly focus on enacting the deployment, the blue entities provide the monitoring (cf. section 4.2.3) and the yellow component enacts the adaptation (cf. section 4.3.2). Clouidiator consists of a **home domain** for which *Colosseum* is the entrypoint offering a JSON-based REST interface. This constitutes the background over a graphical Web-based user interface, but can also be used by adapters and automation tools. It also comprises various Clouidiator internal registries that store information about Clouidiator users, cloud providers, user cloud accounts, and meta-information about cloud offerings such as the operating systems of images. Moreover, the home domain contains a repository of application components together with their life-cycle handlers as well as applications composed of these components. In addition, the internal registries contain information about started VMs and the component instances deployed on them as well as about the wiring between the component instances. Finally, the workers synchronize the internal registries with the cloud provider information, and execute the provisioning of virtual machines or the installation of application components on virtual machines. The *Sword* abstraction layer realises the communication with the various cloud provider APIs based on Apache jclouds.

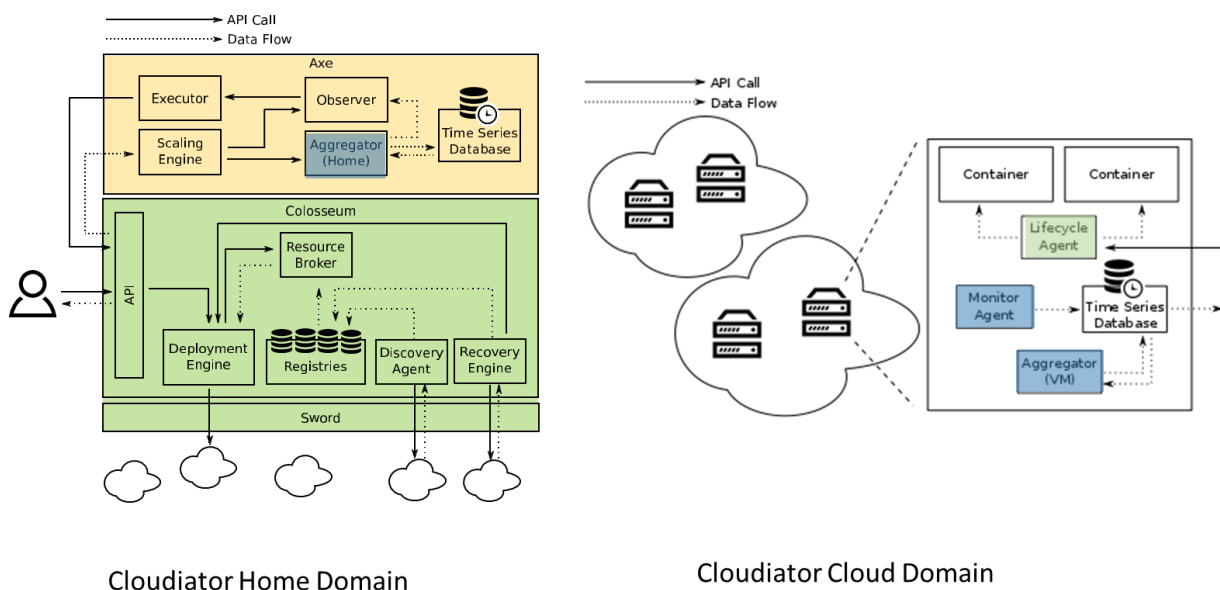


Figure 19 - Clouidiator architecture

³² <https://github.com/clouidiator>

The **cloud domain** comprises all VMs at various cloud providers as well as the component instances running on them. In addition to that, it contains Clodiator's Lifecycle Agent *Lance* on each of the VMs that the home domain uses in order to distribute component instances over VMs and to poll the status of the component instances to ensure a stable deployment. Lance manages the lifecycle of each application component based on an application component description derived from the CAMEL file.

An application component description defines a set of lifecycle handlers that describe how to provision the binaries of the component, how to configure it, and how to run it. Other handlers capture the shutdown of the instance of an application running in the cloud. The lifecycle concept of Clodiator is heavily influenced by Cloudify and CloudML [3]. The lifecycle can be specified as script files, command line instructions, Chef Recipes³³, or Java commands. In addition to that, Clodiator supports two special handlers: The *start detector* serves the purpose of detecting whether an application has started successfully. It is run after the component instance has been started and is used to determine when it is ready for wiring other instances. Once an application is considered running, the *stop detector* is invoked periodically in order to find out whether the application has accidentally stopped. Beside lifecycle handlers, a component description defines open ports that other components can use. Further, it defines ports a component will consume from other components. For both, incoming and outgoing ports, the cardinality of connections can be defined.

4.1.2.2 Evaluation

In the context of CloudSocket, a thorough comparison of existing cloud orchestration tools [48] against the original Clodiator version was performed. This comparison unveils the need for a Clodiator extension to enable the full deployment support for the BPaaS paradigm, which is currently not considered by any of the existing orchestration tools. In addition, the comparison has provided further input to the UULM Monitoring (cf. section 4.2.3) and Adaptation (cf. section 4.3.2) approaches.

The selection of the analyzed tools is based on the particular requirements such as the availability of an open source version, including a documentation and an initial guideline on how to set it up. Further, the selected tools might also be used in other research projects, e.g. Apache Brooklyn in the SeaCloud project. We do not consider tools that by design only support a single cloud platform such as OpenStack Heat³⁴.

This led to four possible tools, namely Scalr, Stratos, Brooklyn, and Cloudify from which Scalr was omitted to favour mere open source projects (Brooklyn and Stratos) and due to the need to support modelling standards (Cloudify) rather than custom formats (Scalr).

The versions of the evaluated tools are *Apache Brooklyn* (version 0.7.0-M2-incubating), *Cloudify* (community edition 3.2), *Apache Stratos* (version 4.1.0-RC2) and *Clodiator* (version 0.1). The comparison is based on feature sets mapping to cloud-related aspects, application-related features and non-functional requirements. An overview of the comparison is provided in Table 3 followed by detailed feature and result description.

³³ <https://www.chef.io/solutions/devops/>

³⁴ <https://wiki.openstack.org/wiki/Heat>

Feature	Tools			
	Brooklyn	Cloudfify	Stratos	Cloudiator
Cloud Features				
Multi-Cloud				
# of Cloud Providers	jclouds	3	jclouds	jclous + n
Abstraction Layer	✓	✗	0	✓
Cross-Cloud	✓	✓	✓	✓
External PaaS	✗	✗	✗	✗
Cloud Standards	✗	✗	✗	✗
BYON	✓	✓	✗	✓
Application Features				
Model Standards	0	0	✗	0
Resource Selection				
Manual Binding	✓	✓	✓	✓
Automatic Binding	0	✗	✗	✓
Dynamic Binding	✗	✗	✗	✗
Life Cycle				
Shell Scripts	✓	✓	✗	✓
# of DevOps Tools	1	3	1	1
Wiring & Workflow				
Attribute & Event	0	✓	✓	✓
Manual Workflow	✗	✓	✓	✗
Automatic Workflow	✗	✓	✗	✓
External Services	✗	✗	✗	✗
Non-functional requirements				
Discovery	✗	✗	✗	✓
Authentication	✓	✓	✓	✓
Multi-tenancy	✓	✗	0	✓
✗ = not fulfilled 0 = partially fulfilled ✓ = partially fulfilled				

Table 3 - Cloud Orchestration Tool Comparison

Multi-Cloud Support Feature: Supporting multiple cloud providers is one of the most crucial features for cloud application management tools, as it allows selecting the best matching cloud offer for an application from a diverse offering landscape. Cloud providers often differ from each other regarding their API. This causes the user to suffer from a vendor lock-in once he depends on the native API of a cloud provider. For that reason cloud, deployment and management tools should offer a cloud abstraction layer that hides differences, avoids the need for provider-specific customisation, and removes the vendor lock-in. Only this feature enables a seamless change of the cloud provider.

Apache Brooklyn uses *Apache jclouds* as cloud abstraction layer and therefore supports many public and private cloud providers. *Cloudify* comes with plugins supporting AWS³⁵, Openstack³⁶ and VMWare vCloud³⁷. It also offers a contributed plugin for Apache Cloudstack³⁸. Nevertheless, *Cloudify* does not support an abstraction layer and each model needs to explicitly reference cloud provider specific features. *Apache Stratos* utilises *jclouds* as a cloud abstraction layer, supporting multiple providers. Yet, the abstraction is imperfect as application specifications still need to refer to cloud specific entities. The *Cloudiator* abstraction layer is built on *jclouds* with additional cloud provider support (e.g. Flexiant Cloud Orchestrator³⁹) provided by UULM. *Cloudiator* allows the cloud provider independent resource specification.

Cross-cloud support Feature: Enhancing the multi-cloud feature such that the user is able to deploy a single application in the way that its component instances are distributed over multiple cloud providers. For instance, the database may be deployed in a private cloud on the user's premises while numerous instances of the application server run in a public cloud. The advantages of cross-cloud deployment are three-fold: (i) It allows a sophisticated per component instance selection of the best-fitting offer; (ii) it enhances the availability of the application as it introduces resilience against the failure of individual cloud providers; (iii) it helps coping with privacy and security issues (private vs. public cloud).

Apache Brooklyn supports cross-cloud deployments on a per-component level: Each component can be bound to a separate cloud provider by referencing its configuration. *Cloudify* offers cross-cloud support. For each virtual machine defined in the model, the user can reference a different cloud provider. *Apache Stratos* allows the definition of network partitions that are logical groups of IaaS resources such as regions or availability zones. Network partitions enable cross-cloud scaling and deployment using policies like round robin through available network partitions. As *Cloudiator* does not link application specific entities with cloud provider specific entities a cross-cloud deployment can easily be achieved, i.e. an application description is completely independent from the underlying cloud provider.

External PaaS Support Feature: In addition to supporting IaaS clouds, the support of PaaS clouds (e.g. Google App Engine⁴⁰) is desirable. PaaS offers ready-to-deploy application containers, thus reducing the complexity compared to IaaS as well as the management effort for the user. On the downside, it comes at the cost of reduced flexibility as the provider defines the container configuration.

None of the four tools allows the usage of external PaaS clouds.

Support of Cloud Standards Feature: In addition to supporting multiple cloud provider APIs, the adoption of cloud API standards such as CIMI [18] and OCCI [56] enables supporting any cloud provider conforming to such standards. None of the four tools supports any cloud API standard.

Bring Your Own Node (BYON): BYON captures the ability to use already running servers for application deployment. It enables the use of servers not managed by a cloud or virtual machines on unsupported cloud providers.

Apache Brooklyn supports BYON by providing an IP address and login credentials for the server. *Cloudify* supports BYON through an externally installable Host-Pool Service that works as a cloud middleware mock-up. When enabled, *Cloudify* requests IP addresses and login credentials from this service whenever it needs to provision a

³⁵ <http://aws.amazon.com>

³⁶ <https://www.openstack.org/>

³⁷ <http://www.vmware.com/de/products/vcloud-suite>

³⁸ <https://cloudstack.apache.org/>

³⁹ <https://www.flexiant.com/flexiant-cloud-orchestrator/>

⁴⁰ <https://cloud.google.com/appengine/>

new server. *Apache Stratos* does not support BYON, despite the general ability of *jclouds* to do so. *Cloudiator* can support BYON with the main requirement that *Lance* (cf. section 4.1.2.1) has to be installed on the server.

Model Standards Feature: Supporting open standards such as TOSCA [57] and CAMP [58] for modelling the application topology, the component life cycles, and the interaction with a cloud management tool facilitates the usage of such tool and further increases the reusability of the topology definition, as it avoids moving the vendor lock-in from the cloud provider level to the management tool. Moreover, it reduces the initial effort and costs to learn a new DSL.

Apache Brooklyn's YAML format follows the CAMP specification, but uses some custom extensions. Yet, it is possible to deploy CAMP YAML plans with Brooklyn and via the separately provided CAMP server. Support for TOSCA is planned for a future release. While *Cloudify*'s DSL for the deployment description is strongly aligned with the TOSCA modelling standard it does not directly reference the standard types, but instead defines its own profile following the TOSCA Simple Profile in YAML [2]. *Apache Stratos* does not implement any standard. *Cloudiator*'s concept does not follow one specific standard but due to the modular approach followed, the support of a specific standard is simple to realise via adapters. Currently CAMEL is supported and for future releases, a TOSCA adapter is planned.

Resource Selection Feature: The resource selection is part of the application topology description. It defines the resources used for the deployment of a component instance in an IaaS cloud. Hence, a resource will commonly refer to the virtual machine type/flavour, an image type, and a provider specific location: `<hardware; image; location>`. A tool has mainly three possibilities to define or derive such a tuple: (i) in a manual binding the user provides the concrete unique identifiers of the cloud entities; (ii) in an automatic binding the user defines abstract requirements regarding the defined tuple (e.g. number of cores). These are then bound to a concrete offer at runtime by the tool; (iii) dynamic binding offers a solving system that enables changes to the binding based on runtime information, e.g., metric data collected from the monitoring system (see section 4.1.3.1).

Apache Brooklyn supports manual as well as basic automatic binding. For the latter it supports resource boundaries for the hardware. The resource selection happens either in the global or in the component-specific parts of the blueprint. *Cloudify* exclusively supports manual binding of the resources used for a virtual machine. The reference to a cloud provider specific node type (e.g. `cloudify.openstack.nodes.Server` for Openstack) has to be defined by the user. Due to this shortcoming, automated and dynamic bindings are also not possible. The resource selection in *Apache Stratos* is a manual process when configuring cartridges by referencing an image and a hardware description in an IaaS cloud. *Cloudiator* supports the manual and automatic binding of resources by providing concrete and abstract description mechanisms. Further more sophisticated resource selection concepts, i.e. facets and generic boundaries, will be enabled in future releases of *Cloudiator*. [59]

Life Cycle Description Feature: The life cycle description defines the actions that need to be executed in order to deploy the application including all its component instances on started virtual machines. The basic approach for the life cycle description of the application is to provide shell scripts that are executed in a specific order. This approach can be extended to support DevOps tools such as Chef that offer a more sophisticated approach to deployment management and ready to use deployment descriptions.

In *Apache Brooklyn* each defined type provides basic life cycle actions called effectors. These can be configured in the concrete application component definition. The configuration can happen either with shell scripts or by referencing Chef Recipes. *Cloudify* relies on the interface definition of TOSCA for defining life cycle actions. The base node type defines multiple life cycle actions as interfaces that are executed during deployment. The actions are defined as shell scripts or by using Chef and Puppet. *Apache Stratos*' life cycle description software setup is delegated to Puppet. *Cloudiator* supports basic shell scripts and the support for Chef Recipes is in process. Further concepts of holistic lifecycle handling is discussed in section 4.1.2.3.

Wiring and Workflows: Most cloud applications are distributed applications where components reside on different virtual machines, e.g., the application server resides on a compute-optimised host, while the database is on a storage-optimised host. Hence, the modelling language needs to support a way to configure those communication relationships between the components by offering a way to pass the endpoint, either before the start of the dependant component (database starts before application server) or after (application server is added to already running load balancer). A straightforward approach to resolve those dependencies is attribute and event passing. That is, the tool allows the user (life cycle scripts) to lock/wait for attributes to become available or register listeners on topology change events. This is commonly achieved by a global registry shared between all component instances of an application. Obviously, this approach offloads most complexity to the user, who needs to, e.g., make sure that the database URL is only available when the database is already started. An improvement is a manual workflow definition. Here, the user defines a workflow taking care of the deployment order. Finally, the easiest way for the end user is an automatic workflow deduction, where the modelling language is sufficiently verbose to allow the system to automatically deduce the correct workflow from the defined life cycle actions on the virtual machines and their relationships.

Apache Brooklyn supports wiring by attribute-and-event-passing. It offers a locking action that waits until the dependent service provides a required attribute. The reverse way, where a later starting service needs to reconfigure a running service, is not supported out of the box. Instead, the user has to implement this functionality. *Apache Brooklyn* supports neither workflow scenarios nor access to external services. *Cloudify* uses the relationship mechanism of TOSCA. It defines a generic relationship type that offers the execution of custom actions on either the source or the target of the relationship on specific events. Combined with a shared configuration space available via, e.g., a shell extension, this allows the user to configure endpoints before or after the start of a service. The user can implement custom workflows, making sure that the life cycle actions are executed in the correct order. If the user only uses the basic life cycle actions, *Cloudify* is capable of automatically deducing the correct execution order. *Cloudify* does not support external services by default. *Cloudiator* uses an extended approach of attribute-and-event-passing that supports the reconfiguration of running services if services are added later at runtime. In order to ease the deployment for the user, *Cloudiator* automatically derives the deployment workflow from the modelled communications between the services. Therefore a manual deployment workflow is not supported. The support of external services is currently not supported by *Cloudiator* but as outlined in section 4.1.2.3 this is a planned feature for the next release.

Discovery Feature: Discovery means that the given tool is able to automatically retrieve the different offerings such as images, hardware flavours and locations from the cloud providers. Having the different offers directly discovered by the system is beneficial to the user: It reduces the initial effort of “manual” discovery and it is less prone to errors such as typos. Moreover, it can be kept up to date automatically.

Apache Brooklyn, *Cloudify* and *Apache Stratos* do not support automatic discovery. Hence, the user has to set cloud-specific unique identifiers by hand. *Cloudiator* retrieves images, flavours and locations automatically and updates them in its local registries in case the state changes at the cloud provider.

Authentication and Authorisation: The cloud management tool offers a single point of attack. It stores cloud provider authentication information and entirely controls the application. This offers the possibility to, e.g., shutdown or even delete the application, but also to access the virtual machines. To protect sensitive information, the tool should at least offer authentication. As multiple persons in general maintain an application, it should also offer a multitenancy mechanism. Finally, a fine-grained authorisation mechanism, allows defining roles within the system, giving different privileges to different users. For instance, this allows that only a limited set of persons can shut down the application, while others can only retrieve monitoring reports.

Apache Brooklyn supports authentication for the dashboard and its REST API being enabled by default. Multi-tenancy and authorisation have been recently integrated via the Entitlement Manager. While *Cloudify* has all

security features disabled by default, it offers ways to secure the communication with its manager. The REST API of the manager can be secured by either password or token based authentication, securing the access via command line and Web GUI. Multi-tenancy is currently not supported. *Apache Stratos* provides users, roles and tenants through the Web GUI and the API. Moreover, the credentials of the underlying used IaaS cloud provider are only stored in configuration files and are not visible, or editable, directly through Stratos. Authentication and Authorisation in *Apache Stratos* exists, but presumably during development some features were disabled (e.g. multi-tenancy, roles etc.). *Cloudiator* supports authentication for the dashboard and its REST API, both enabled by default. Further, *Cloudiator* supports multi-tenancy.

4.1.2.3 Extension: PaaS orchestration and abstraction layer

As seen before, PaaS abstraction is still an open and on-going topic in current research. It can drastically decrease costs, by e.g. sharing infrastructure of public PaaS providers to host an application. Therefore, the integration of a PaaS abstraction into the Cloud Provider Engine is of major interest for CloudSocket.

Relying on our use case of a card designer of the Greeting Cards BPaaS Bundle, we can decrease the cost of the deployment, since the cost of a virtual machine (IaaS level) per hour is higher than the cost of an application server (PaaS level). Therefore, in this use case, we can run the WAR file, i.e. the actual application, on an Apache Tomcat of a public PaaS provider instead of the IaaS provider. Of course, this comes with several restrictions, such as the limited means of monitoring and configuring the actual service.

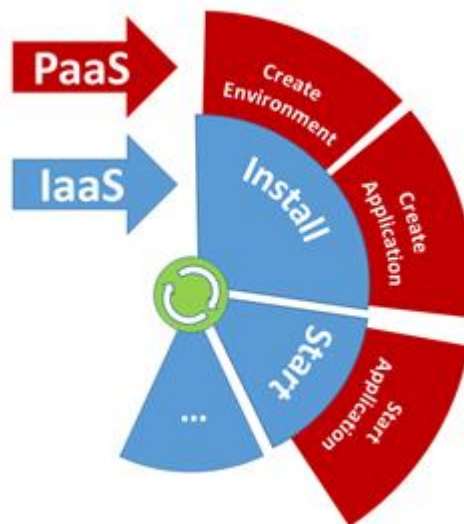


Figure 20 - Unified Life-cycle Handling in the Cloud Provider Engine

In the current Cloud Provider Engine, there is already an abstraction layer for IaaS providers called Sword. Our approach is (i) to extend the state machine of the life-cycle management to cover also PaaS-specific actions and events, such as the creation of an environment, and (ii) to overload the description of the components as is described in section 2.2.3.2). This is enabled by using application manifests, instead of e.g. deployment scripts. Figure 20 shows an excerpt of the unified cross-layer life-cycle for Cloud providers.

We call this PaaS abstraction layer Dagger. The life-cycle management of Colosseum is then capable of executing the respective calls in a provider-agnostic way as it is in Sword. Figure 21 shows the architecture of the Cloud Provider Engine after the integration.

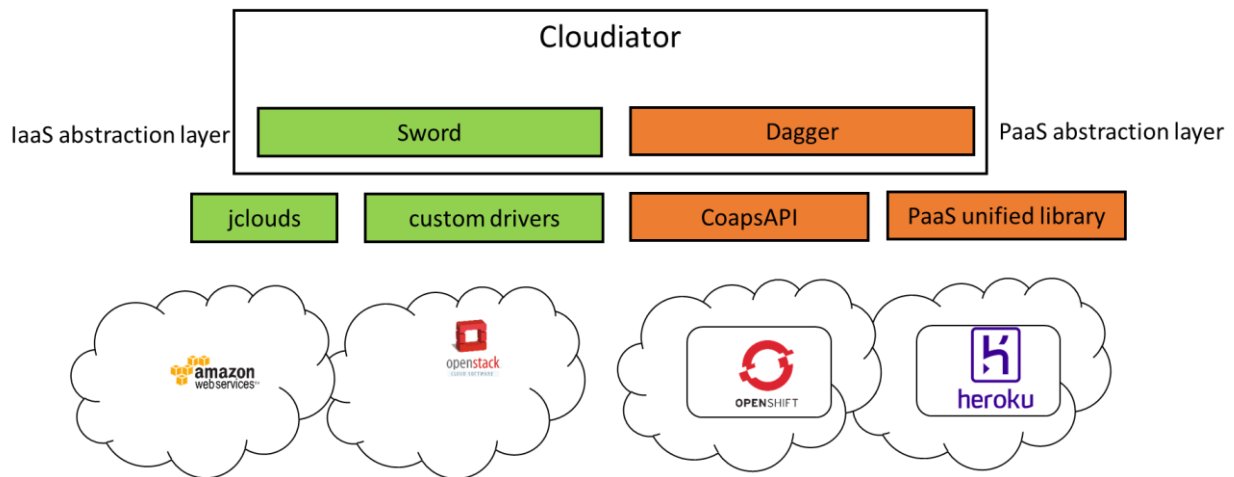


Figure 21 - Cloudiator with IaaS and PaaS abstraction layer

As Sword heavily relies on jClouds to implement calls, we target on the integration of one of the following tools to be used in the PaaS abstraction layer: (i) the PaaS Unified Library or (ii) the COAPS API.

The PaaS Unified Library⁴¹ from the SeaClouds EU project is a library that provides simple operations for managing applications in PaaS providers: deploy, undeploy, start, stop, scale and bind service. It also includes an optional REST interface on top of the library, allowing the use of the library as a standalone application. The current supported providers are CloudFoundry v2 based providers, e.g. Pivotal, Bluemix or Canopy Cloud Fabric, OpenShift v2 based providers, e.g. OpenShift Online, and Heroku. It relies on the official Java clients for each platform. The CloudFoundry and Heroku implementations are able to deploy supplied artefacts, while the OpenShift implementation requires a URL to a git repository. Finally, Heroku is restricted to Java web applications. The Life-cycle part of a specific application is implemented in the methods to start, stop and remove an application. Scaling is done via an additional methods that allows to change the amount of instances that are associated to a module. Additionally, the CloudFoundry implementation is able to scale disk and RAM. The values have to be provided by the user. A service management API allows binding existing services to an application where applicable. The PaaS Unified Library does not use a unique credentials for each cloud provider. On the contrary, each library session needs the credentials to be provided by the user. This allows the implementation of a service that is able to manage PaaS applications from multiple users, but it is less convenient if this service is supposed to act as the PaaS broker.

The (M-)COAPS API^{42,43} is an specification for an abstraction interface of common PaaS provider deployment APIs. It comes as an independent application that is run in an application server and provides a REST -ful API in terms of a proxy for several PaaS platforms. The supported providers are currently OpenShift, Amazon Elastic BeansTalk and Cloud Foundry. Implementations for the Google App Engine and Appscale are currently under development. The core life-cycle actions are very similar to the ones of the PaaS Unified Library. Table 4 shows the common methods of both APIs. In addition, the Unified PaaS Library has some more specific methods for the management of users, service bindings, security policy, scaling, monitoring and so on that are not available throughout all PaaS providers and therefore not considered in our very general approach for integrating a PaaS abstraction. Scaling is not explicitly as own methods in the COAPS API, but it is possible to change the number of instances (horizontal scaling) and the description of the environment (vertical scaling) in terms of updating the specific entities. In addition, service management, in respect of binding services of given providers to the application, is not an integral part of the COAPS API. However, as this is always application specific, the credentials to the service might be

⁴¹ <https://github.com/SeaCloudsEU/unified-paas>

⁴² <http://www-inf.it-sudparis.eu/SIMBAD/tools/COAPS/>

⁴³ The specification is released as COAPS API and later implemented and continued as M-COAPS API.

passed via environment variables. The credentials are not managed dynamically, but have to be provided in a config file, that can be changed throughout instantiations of the COAPS API application.

PaaS Unified Library	COAPS API
Create Container	Creating Environment
Update Container	Updating Environment
Delete Container	Destroying Environment
	Finding Environments
	Describing Environment
Retrieve Container	Getting Information
	Getting Deployed Applications
Create Application	Creating Application
Update Application	Updating Application
Get List of Applications	Finding Applications
Manage Application	Starting Application
	Stopping Application
	Restarting Application
Retrieve Application	Describing Application
Delete Application	Destroying Application
	Destroying Applications
Create Deployment	Deploying Application
Delete Deployment	Undeploying Application

Table 4 - Life-cycle Actions of the Generic PaaS Deployment APIs

The outcome of the analysis of these APIs is that we target for the Cloud Provider Engine, the integration of the methods of the COAPS API in our life-cycle actions. This means the current abstraction layer, formerly solely represented by Sword, is extended by another component called Dagger that handles the communication with the PaaS providers. The state machine of the lifecycle management extended to cover all possible states; even some of them only needs triggering when a PaaS-layered component is involved. By that, when the state of the component is at *install*, it will call the *install*-method for IaaS-based components and in the case of PaaS-based components it will enact the following chain of commands: create environment, create application, deploy application.

In respect of the final integration in the Cloud Provider Engine, there are several implementation aspects to be considered. The COAPS-API is stand-alone applications that runs independently of the Cloud Provider Engine in

an application server and is accessed by a REST interface. The PaaS Unified Library also offers a REST façade and it supports the usage as a plain library. By a prototypical implementation that is currently under development, we check the feasibility of the integration of the COAPS API into the Cloudiator framework. If this would not work as needed, we consider the plain integration of the methods in a novel abstraction layer with the abilities of the Cloudiator framework. As was already proven in the IaaS abstraction, the framework provides a rich set of features that handle the communication with and the management of Cloud providers in a stable and reliable way. By the means of the watchdog system, the advanced and sophisticated scheduling and registry components, we most likely achieve better results by a native integration. However, to proof the feasibility of integrating this cross-layer life-cycle, the first step is the integration of the COAPS API.

Further, on, we extend the meta model of Colosseum, i.e. the main part of the Cloudiator framework, to allow instances to be run on containers (i.e. environment in the notation of the COAPS API). The description of containers features attributes, also found in the flavour part of the virtual machine entity, like RAM and Storage, but with additional fields, we extract from the environment manifest. In this manifest, configurations like the runtime of the environment (e.g. Java 7) is defined

4.1.3 Future Research

4.1.3.1 *Dynamic IaaS Selection at Runtime*

While a deployment plan might explicate deliberately which IaaS services are going to be instantiated and host respective BPaaS internal service components, there can be scenarios that such a plan is not complete. For instance, imagine that one IaaS offering is available in multiple cloud locations. In this respect, we might desire to select the offering instance, which is more close to the BPaaS client in order to reduce the overall execution time of the BPaaS workflow. To this end, in order to cover such more dynamic scenarios, we plan to investigate a more dynamic IaaS selection approach where the specific VM offering or even the actual cloud location for such an offering is selected. To enable such type of selection, we could rely on different types of criteria. One criterion, already mentioned, could be the location with respect to the BPaaS client. Another criterion could consider the previous performance of the respective component that needs to be hosted in the respective cloud. As such, one issue that has to be dealt with concerns the suitable determination of the criteria that need to be used in the selection and can cover the dynamic scenarios that we are aiming at. The second issue is how to perform the IaaS selection. We can rely on a QoS-aware service matchmaking approach like the one mentioned in section 3.2.1.1. This is because this kind of selection includes the matchmaking of constraint-based specifications over functional properties of the IaaS offering as well as non-functional terms (quality metrics and attributes). Above all, of course, we also need to define exactly the dynamic scenarios that we need to cover. Apart from endpoint/cloud location selection, we can also imagine that we might require to dynamically selecting a cloud offering at runtime in order to address prospective problems like offerings not existing any more or offerings that have become problematic due to various reasons. By relying on a static selection approach, such problems cannot be really addressed and can lead to the need of performing adaptations at the IaaS level in order to properly confront them resulting in lost time and increased cost. Based on the above analysis, the dynamic IaaS selection at runtime is considered as an interesting research direction that could be followed in the forthcoming project period.

4.2 Monitoring

Monitoring of a BPaaS or any kind of service is crucial for evaluating the performance of this service and be able to adapt it in case deviations from the expected performance behaviour are detected. To this end, the respective mechanisms and metrics should be in place in order to realise such monitoring by being coupled also by a

corresponding distributed monitoring framework. Distribution is necessary in order to handle an increased monitoring load especially originating from the lower layers of abstraction as well as to be able to overcome failures, thus paving to address issues concerning single points of failure. The metrics to be measured should be carefully designed in order to cover all possible requirements that can be posed on the different layers. Such metrics could be drawn from a repository of widely used and common metrics (i.e., the Metric Registry) while new metrics might need to be specified. In any case, the monitoring framework should be able to measure all the respective metrics needed and be extensible to address the needs of incorporating new metrics on demand. In addition, it might be required that such a framework is automatically reconfigured in order to address unforeseen load or modifications to corresponding requirements or the unexpected failure of some monitoring nodes. Finally, to address cross-layer adaptation (see section 4.3) and ensure the computability of all the metrics required, the monitoring framework should be able to propagate and aggregate measurement information across different layers by exploiting metric models that cover the dependencies between the different layers. Such metric models could take the form of metric trees, which explicate the way low-level measurements can be propagated up to the highest level producing the respective measurements of metrics at that level.

4.2.1 State of the art

In the scope of CloudSocket the monitoring applies to the three areas: (i) Quality Models, covering the monitoring of QoS terms in general, (ii) Service Monitoring, covering the monitoring of services and service-based applications and (iii) Cloud Monitoring with the focus on the cloud related monitoring challenges.

4.2.1.1 Quality Models

Various quality models have been proposed in the literature. They can be distinguished in general to those that are layer-specific, thus focusing on one layer, or cross-layer, focusing on two or more layers. Quality models can also be separated into those covering one or several aspects. In particular, we can see in the literature quality models that focus on security, scalability or elasticity aspects or models that attempt to cover them all. Based on the survey in [16], the quality models can be evaluated across different criteria. These criteria include the extensiveness/richness of the model, the coverage of both provider and requester views, the coverage of domain-independent and dependent quality terms, the coverage of QoS and QoE quality terms, the association of attributes to metrics from which they can be computed, the layers covered, and the type of quality term dependencies captured. The analysis in this survey revealed that there is a trend towards providing more rich models which tend to cover many layers and not just one as well as to cover most of the comparison/evaluation criteria. Our analysis now focuses shortly on reviewing some layer-specific and cross-layer (cloud-based) quality models.

The quality models in [60], [61] focus mainly on the service and infrastructure layers but not on cloud services. They actually cover mainly the service provider view and domain-independent QoS metrics. They do provide a mapping between quality attributes and metrics while the structuring of these models can be considered to be at a good level with moderate or good detail level.

Cross-layer quality metrics have been proposed in [39], [62]. Compared to the previous quality models, they are better and cover both provider and requester views, both QoS and QoE terms as well as map to a high level of detail. Some trade-offs exist between these models. The model in [63] has a better detail level than the one in [64] but does not cover dependencies. On the other hand, the model in [62] has the best performance in all criteria but does not cover the specification of metrics and respective computation formulas.

The CLOUDQUAL quality model has been proposed in [65] which covers dimensions and metrics for cloud services in general. This model comprises 6 well-known main quality dimensions: usability, reliability, availability, responsiveness, security and elasticity. However, the coverage of metrics mapping to these dimensions is quite low where a one-to-one mapping between dimensions and metrics is captured.

In [66] a cloud-based quality model is proposed which focuses on the coverage of non-functional, economic and technical aspects. For each aspect, a different set of dimensions and respective metrics are specified. This quality model seems to cover all service layers. However, we believe that the categorisation of the quality terms is more or less inappropriate and some quality aspects like scalability, security and elasticity are not well-covered.

Li et al. [67] have performed a survey over metrics that can be used to evaluate cloud services and have come up with a quite extensive quality model. The quality dimensions covered are partitioned into two main parts: the physical and the capacity one. The physical one includes the dimensions of communication, computation, memory and storage, while the capacity one includes the dimensions of transaction speed, availability, latency, scalability, reliability, variability and data throughput. In addition, the authors show the dependencies between the properties in each part and across these two parts. Furthermore, the proposed model also covers economic, elasticity and security dimensions, attributes and metrics. For each quality dimension covered, multiple quality metrics are defined. However, many of them are benchmark-oriented and cannot be directly used by a monitoring system.

The SMI index is proposed in [68] which can be used to express KPIs and enable their respective assessment over cloud-based services. This SMI index focuses on 8 main quality dimensions: accountability, agility, assurance of service, cost, performance, security, privacy and usability for which 13 quality attributes are defined, covering response time, accuracy, availability, reliability, stability and cost and elasticity. For each quality attribute, few quality metrics are defined. For instance, for response time, we have average and maximum response time as well as response time failure. The quality model proposed seems generic enough to cover the 3 main layers in the cloud abstraction stack but does not provide enough details for particular dimensions and attributes.

Various research approaches have focused on defining meaningful scalability and elasticity metrics as well as providing formal definitions for these two quality terms. In [69], the authors distinguish between application and platform scalability. Application scalability bears on an application's ability to sustain particular performance levels when its workload increases, while platform scalability is similarly defined as the ability of the execution platform to provide as many resources as needed. On the other hand, elasticity is defined as the degree to which a system can dynamically provision and de-provision resources in an autonomic manner to cover as closely as possible the current demand. Then, the authors defined two elasticity dimensions, namely speed and precision that were mapped to two main metrics to measure them. In [70], a new elasticity metric is defined which covers well the aspects of scalability, accuracy, time and cost and is computed from a function which takes into account these 4 aspects. The function includes more simple metrics which can be computed from SLA and/or historical information. The authors in [71] evaluate elasticity as a financial penalty related to the under- or over-provisioning of resources in the context of the cloud service customers. The survey in [72] reviews various approaches focusing on the definition of quality metrics covering scalability, elasticity and cloud service efficiency. The approaches are evaluated according to whether they cover all three aspects, whether they cater for the service provider or requester view and whether they cover the three main layers in the cloud computing stack. Another criterion for evaluation is the consideration/coverage of different scalability concepts. Finally, the authors in [73] follow the goal-question-metric approach in order to come up with metrics measuring cloud service scalability, elasticity and efficiency. The final quality model produced includes two metrics for scalability, namely speed and range, two metrics for elasticity, namely mean-time-to-quality-repair (MTQR) and number of SLO violations, and two metrics for efficiency, namely resource provisioning efficiency and marginal cost.

Concerning the workflow layer (WfaaS), one of the most prominent work is the one in [11] which defines a quality model that covers both the workflow and task level. This quality model considers three main dimensions: time, cost and reliability and defines respective quality metrics for them. The metrics defined for the task and workflow level are equivalent but the main intuition is that the workflow-level metrics can be derived from the task-level ones by also considering the workflow structure. For the time dimension, the parent/root metric is task response time, which is broken down into a tree of more simple metrics where at the second level we have the task process and delay time. The task cost is the root metric in a shallow metric tree hierarchy where in the second level the total task cost

is split into the enactment and the realisation cost. Concerning reliability, two main metrics were defined: system failure rate and process failure rate.

An interesting quality model which focuses mainly on properties and aspects that need to be measured but not metrics has been proposed in [74]. This quality model seems to apply mostly on the IaaS and possibly PaaS layers and includes the aspects of scalability, elasticity, reliability, adaptability, timeliness, autonomy, comprehensiveness and accuracy. This quality model captures those properties that have to be exhibited by a monitoring system in the cloud. Thus, this is yet another perspective with respect to the usual one that concentrates on capturing the service provider and/or requester views.

4.2.1.2 Service Monitoring

In this sub-section, we present research work that has focused mainly on the monitoring of services. This work is not accompanied by a service adaptation sub-system. On the other hand, in section 4.3.1, we will also analyse approaches that are able to both monitor and adapt services.

[75] presents an event-based monitoring approach, developed within the Astro project, which also extends the ActiveBPEL engine and defines RTML, an executable monitoring language to specify SBA properties. Events are combined by exploiting past-time temporal logics and statistical functions. Monitors run in parallel with the BPEL process as independent software modules verifying the guarantee terms by intercepting the input or output messages received or sent by the process. This work does not allow for dynamic (re-)configuration of the monitoring system in terms of rules and meta-level parameters.

In [76] the authors present an approach towards extending WS-Agreement. This approach supports monitoring of functional and non-functional properties. EC-Assertion is introduced to specify service guarantees in terms of different types of events, which are defined in a separate XML schema and it is based on Event Calculus (EC). By proceeding in parallel with the business process execution, it leads not only to less impact on performance, but also to a smaller degree of responsiveness in discovering erroneous situations.

A platform for developing, deploying and executing SBAs is proposed in [77], incorporating tools and facilities for checking, monitoring and enforcing service requirements expressed in WS-Policy notations. The Colombo platform comes with a module that manages policy assertions. Apart from evaluating the assertions attached to particular service-related entities at both design and run-time phases, the framework provides the means for policy enforcement, e.g., it may approve a delivery of a message, a rejection of it, or defer further processing.

4.2.1.3 Cloud Monitoring

Lifting monitoring to the cloud comes along with various requirements compared to traditional server monitoring [74]. The monitoring of cloud stack encompasses different layers that need to be monitored, i.e., operating system, middleware (PaaS) and the actual application. Further requirements defined by [x] are scalability and elasticity, i.e., the monitoring system has to handle a large number of probes and has to cope with dynamic changes of the monitored entities. Tools provided by cloud providers, such as Amazon's CloudWatch⁴⁴ or CloudMonix⁴⁵ suffer from vendor lock-in. In addition, further tools are required when data from different cloud providers shall be aggregated. Established open source monitoring tools such as Ganglia⁴⁶ or Nagios⁴⁷ are designed to monitor large distributed systems, but struggle with the dynamicity of cloud environments. More cloud-aware monitoring systems such as DARGOS [78] offer a scalable architecture with the focus on OS and customisable application specific metrics. Additional service levels like PaaS or SaaS are not considered in the DRAGOS approach. Further, cross-

⁴⁴ <http://aws.amazon.com/en/cloudwatch/>

⁴⁵ <http://cloudmonix.com/>

⁴⁶ <http://ganglia.sourceforge.net/>

⁴⁷ <https://www.nagios.org/>

cloud monitoring and respective challenges are not addressed at all. The PCMONS [79] approach focuses on private cloud infrastructures with an extensible architecture to overcome the vendor lock-in on the IaaS level. Higher-level as well as cross-cloud monitoring is not considered by PCMONS. The monitoring system proposed by König et. al. [80] targets the monitoring of all cloud service levels by combining service layer specific solutions in an integrated monitoring system. Its architecture is based on a peer-to-peer system to provide scalability and offers a set of aggregation levels for the gathered monitoring data. This aggregation is provided in a static way and does not allow to be changed dynamically at run-time.

The realisation of a scalable and elastic cloud monitoring system also depends on the applied storage backend of the monitoring systems. With the evolution of NoSQL databases and their scalability capabilities [81] which constitute a widely discussed topic in academia, various monitoring solutions rely on such NoSQL databases for performance and scalability reasons [82]. In the context of NoSQL databases and their usage for monitoring systems, a more monitoring centric database type evolved in the recent years, i.e., the time-series database (TSDB). TSDBs typically build upon a NoSQL database and add further monitoring related functionalities like statistic-based queries, aggregation capabilities and a monitoring optimised data structure [83]. Typical open source TSDB representatives are KairosDB⁴⁸, OpenTSDB⁴⁹, InfluxDB⁵⁰ and Druid⁵¹.

Tower4Clouds⁵², developed in the context of MODAClouds European project is a monitoring platform able to monitor multi-clouds applications. Its main capabilities are: (1) the user defines the QoS constraints, in the form of monitoring rules, which need to be assessed at runtime. These rules are cloud-provider independent; (2) the Data Collectors that are deployed together with the application send the monitoring data to a central Data Analyzer, according to the installed monitoring rules, which specify what and how resources should be monitored. No reconfiguration is required after scaling or migration activities. For PaaS applications, an application data collector is implemented, which is able to collect response times and throughput measurements; (3) The Data Analyzer processes the data gathered by the data collectors, performing aggregations and/or verifying conditions as specified in the monitoring rules. Some predefined actions can be defined in a monitoring rule and executed when a condition is satisfied. Tower4Clouds is the monitoring platform selected in the SeaClouds project. The SeaClouds platform is in charge of automatically deploying the data collectors and configure them once the application is deployed. An additional data collector was developed to measure the availability of PaaS applications; it is an external module, deployed along with the Tower4Clouds platform.

A survey paper, which evaluates many cloud monitoring solutions, both proprietary and open-source, can be found in [74]. These solutions are evaluated based on the quality model that was referenced in section 4.2.1.1 covering the monitoring system performance/quality. This paper provides a nice conceptualisation of the monitoring research problem by indicating 4 main aspects applying to it: (a) the need for monitoring; (b) the basic concepts; (c) the properties to be measured; (d) open issues and future directions. Concerning the latter aspect, we can clearly see some directions that are considered quite relevant with respect to the research that we intend to perform on cloud monitoring. These highly-research directions include: (a) cross-layer monitoring; (b) monitoring of federated clouds; (c) effectiveness; (d) efficiency.

In [84] a monitoring data distribution architecture is proposed which enables cross-site compatibility through the employment of semantic annotations. In particular, semantic annotations are used for the lifting of the monitoring information drawn from different monitoring sources. The ending result is a distributed semantic repository providing SPARQL endpoint via which SPARQL queries can be posed on the semantically lifted monitoring data. Such semantically lifted data are also distributed to potential subscribers via a distribution hub. A nice feature of the latter

⁴⁸ <https://kairosdb.github.io/>

⁴⁹ <http://opentsdb.net/>

⁵⁰ <https://influxdata.com/>

⁵¹ <http://druid.io/>

⁵² <http://deib-polimi.github.io/tower4clouds/>

hub maps to its ability to properly distribute the semantic data according to their type (public/private) thus satisfying respective data policies.

The authors in [85] propose the CASVID monitoring architecture, which stands for Cloud Application SLA Violation Detection and focuses on supporting not only infrastructural but also application-level monitoring. However, connections/dependencies between different-level metrics are not actually considered and thus layer-specific monitoring is actually supported. An interesting aspect of the respective monitoring system proposed is that it enables the automatic detection of the monitoring/measurement interval through the application of a novel algorithm that can be exploited by cloud providers in an individual basis to detect this interval for each application that exploits their services. This algorithm considers different sampling intervals until the point of convergence in the provider's utility which results into the respective interval to be selected.

In [86] a fine-grained cloud monitoring solution is proposed which relies on an in-network switch design (by employing a low-complexity encoding scheme) in order to compress at the network level the monitoring data (mainly status information) that are exchanged. As a proof of concept, the authors highlight the ability of their monitoring solution to early detect stragglers. The authors conclude by indicating that the switches complying to the proposed design can constitute a compressed status information place to be exploited for both the application and infrastructure-level monitoring.

The authors in [65] propose a combined push and pull model for cloud computing monitoring which intelligently switches from one individual model to another one based on user requirements and monitored resources status. The authors claim that this combined model leads to better monitoring performance and caters for different privileges and access styles for the virtualised resources to be monitored. Concerning the latter advantage, the authors also indicate different types of components and how they more efficiently map to one of the models or the combined one.

In [87] a window-based state monitoring framework for cloud applications is proposed which is more robust to value bursts and outliers and follows a respective distributed architecture with two main versions. In the first version, centralised parameter tuning is supported while, in the second version, a decentralised one which enables the monitoring system to scale to multiple monitoring nodes as these nodes rely on the local information to tune their parameters. Two optimisation techniques are also introduced which enable to reduce the communication cost between a coordinator and its monitoring nodes: the first enhances the effectiveness of the global push procedure at the coordinator side while the second one targets the reduction of unnecessary global polls through enabling the performance of local polls when needed.

A framework for collecting application-level measurements is proposed in [88] which exploits the Complex Event Processing (CEP) paradigm. This framework caters for the proper mapping of metrics to event streams as well as their correlation to enable the computation of aggregated measurements mapping to complex metrics. While a simplified monitoring architecture is proposed with no mentioning of how it can be distributed, an interesting event hierarchy is proposed via which correlation can be achieved at the levels of host, resource pool and metric.

A service for estimating, monitoring and analysing cost for scientific cloud-based applications is proposed in [60]. In this service, different cost models are associated to different application execution models and some of the models are combined in order to produce cost for advanced scenarios. This service exploits various techniques to measure cost for scientific applications which also take into account application component dependencies.

In [89] a runtime model for cloud monitoring is proposed that concentrates on common monitoring concerns. Through this model, monitoring data are collected via various techniques and used to construct the performance profile of a cloud. Based on the proposed runtime model, a distributed monitoring framework has been developed with centralised collection/aggregation capabilities which addresses the trade-off between monitoring

overhead/load and monitoring capability via adaptively managing the cloud facilities. This monitoring framework and model seem to be able to cover different levels in the cloud abstraction stack going up to the application level.

4.2.2 Scalability / Elasticity Evaluation of Distributed Databases

Typically, cloud services are on-demand, highly distributed, elastic and scalable. Therefore, the monitoring systems has to cope with these characteristics, leading to similar requirements for the storage backend of the monitoring engine. In the context of CloudSocket, an increasing amount of deployed BPaaS Bundles with scalable services will also increase the amount of monitoring sensors accessing the monitoring engine. Hence, in the context of cloud monitoring, the storage and processing of time-series data becomes one of the common challenges. As the monitoring engine itself is a cloud service, the advantages of the cloud, i.e., scalability and elasticity, should be highly applied.

Especially in the context of cloud computing and distributed databases, the semantics of scalability and elasticity need a more detailed explanation. Following the definition in [90] *scalability* means support for huge datasets and very high request rates. As cloud systems are architected to scale-out, large scale is achieved using large numbers of commodity servers, each running copies of the database software, i.e., database nodes. *Elasticity* builds upon scalability, that provides the ability to have large scale systems. Elasticity means that you can add more capacity to a running system by deploying new instances of each component, i.e. database nodes, and shifting load to them.

Traditional databases like Relational Database Management Systems (RDBMS) were originally designed to provide high performance and consistency in a centralised setup [67]. A new database category came up with the evolution of NoSQL databases that promise to be ready for the cloud by providing scalability and elasticity in a distributed setup [91]. In contrast to RDMS, NoSQL databases do not rely on a fixed schema but focus on performance while offering a less strict consistency. Building upon NoSQL databases as storage backend another category of databases evolved in the last years, the *time series databases (TSDB)* [83], [92]. TSDBs store sets of large monitoring data, i.e., time series data and provide enhanced aggregation operators and a graphical visualisation of the time series data.

Gaining a common knowledge over the scalability and elasticity capabilities of NoSQL databases will lead to producing/developing TSDBs which constitute the most suitable scalable and elastic storage backend for the monitoring engine. Further, this database scalability/elasticity knowledge can be integrated in further environments, e.g., database aware adaptation rules or by enriching service descriptions with non-functional specifications of database scalability ratings. As a starting point three commonly used NoSQL databases⁵³, namely Apache Cassandra⁵⁴, Couchbase⁵⁵ and MongoDB⁵⁶, were benchmarked in the OpenStack infrastructure in order to evaluate their capabilities as storage backend for TSDBs.

4.2.2.1 Evaluated NoSQL databases

As NoSQL databases can be categorized in four different groups, namely key-value databases, document oriented databases, column family databases and graph databases [93]. In the focus of a TSDB storage backend, graph databases are not considered due to their data structure. A short description of each evaluated database is provided.

4.2.2.1.1 Apache Cassandra

Apache Cassandra belongs to the column family databases. It has column groups, updates are cached in memory and then flushed to disk, and the disk representation is periodically compacted. Data partitioning and replication

⁵³ <http://db-engines.com/en/ranking>

⁵⁴ <http://cassandra.apache.org/>

⁵⁵ <http://www.couchbase.com/>

⁵⁶ <https://www.mongodb.com/>

are supported. The architecture of Apache Cassandra is built on the multi-master paradigm that is inspired by peer-to-peer systems [94]. Therefore, each Apache Cassandra node is equal, handling all type of operations. With this architecture, Apache Cassandra promises linear scalability on commodity hardware or in the cloud⁵⁷. The evaluated version of Apache Cassandra is 2.2.6.

4.2.2.1.2 Couchbase

Couchbase belongs to the document oriented databases; however, the usage as key value store is also possible. Couchbase relies heavily on in memory caching as it uses Memcached⁵⁸ before the data is asynchronously persisted to disk. As Apache Cassandra, the distribution architecture of Couchbase follows the multi master paradigm. The evaluated version of Couchbase is 4.0.0 community edition.

4.2.2.1.3 MongoDB

MongoDB belongs to the document oriented databases. While former releases of MongoDB relied on caching only the index in memory and persisting data synchronously to disk, the new 3.X release changed to an extended caching mechanism and asynchronous persistence to disk. In contrary to the architecture of Apache Cassandra and Couchbase, MongoDB relies on three different node types: (1) *Router nodes* act as endpoints for the clients, processing their requests; (2) *Config Server nodes* store the metadata of the cluster. The Router node retrieves the actual location of the request dataset from the Config Server node. (3) The actual datasets are stored at *Shard nodes*. The evaluated version of MongoDB is 3.2.0.

4.2.2.2 Benchmarking Tool

The de facto standard tool in academia and industry for benchmarking NoSQL databases is the *Yahoo Cloud Serving Benchmark*⁵⁹ (YCSB) [95] which was originally developed for benchmarking Yahoo's own PNUTS database [96] and compare it with existing NoSQL databases.

Figure 22 shows the modular architecture of the YCSB where the *Workload Executor* and *DB Adapter* can easily be modified or even replaced by custom modules. The *Client Threads* and *Statistics* modules are the core components of the YCSB and offer various configuration options. As input, the YCSB requires a workload file, describing the actual workload to execute. YCSB supports the CRUD (create, read, update, delete) operations per default and can be extended for operations that are more complex. A workload file contains the actual number of records, the number of operations to execute a distribution of CRUD operations per workload and an algorithm for the record selection probability. In order to achieve comparable results we relied on these basic operations as more complex operations might depend on the actual query implementation of the respective database. The YCSB allows running multiple YCSB in parallel to generate an arbitrary amount of load. The benchmarks were performed with a custom version⁶⁰ of the original version of YCSB release 0.8. The custom version includes updated client drivers for the selected databases.

⁵⁷ <http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html>

⁵⁸ <https://memcached.org/>

⁵⁹ <https://github.com/brianfrankcooper/YCSB>

⁶⁰ <https://github.com/sevbi87/YCSB>

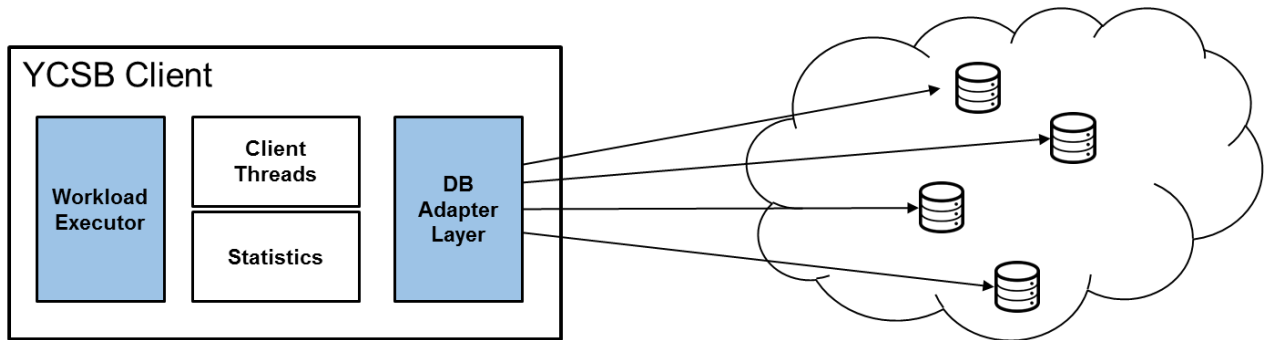


Figure 22 - YCSB Architecture

4.2.2.3 Methodology

Theoretical evaluation and actual benchmarks have been performed in order to evaluate the performance and scalability of distributed databases. [97] benchmarks different NoSQL databases with main focus on the performance by measuring the overall execution time of a particular workload. However, the evaluation is performed in an artificial environment without considering the cloud context. A more on scalability-oriented approach is presented in [98] Different Apache Cassandra cluster sizes are evaluated by measuring again the execution time for different workload sizes. The results show that Apache Cassandra scales as the execution time reduces with an increasing amount of nodes. The cloud context is taken into account by [99] by distributing databases across a cloud infrastructure. Their benchmarks used different VM configurations to analyse the possible influence on throughput and latency. The results led to a first version of a database scalability model.

The elasticity aspect is not deeply investigated in the outlined academic publications and also in industry [97]. Typically, the benchmark setup encompasses static database cluster configurations where the workload is applied. In the context of cloud computing also the elastic provisioning of resources, i.e., database nodes, requires a thorough evaluation of the elasticity capabilities of distributed databases. More precisely the elasticity is defined by adding nodes to a database cluster during (workload) runtime.

The actual benchmarking methodology comprises two setups: (1) a static database cluster configurations, benchmarked by two YCSB clients. The static cluster configuration for each database is alternatively formulated by a 1-node cluster, a 2-node cluster and a 4-node cluster. This approach will provide the basic scalability results by comparing the average throughput (operations per second) for each configuration. In addition, it will provide a general performance comparison between the evaluated databases. (2) The elasticity will be benchmarked by producing a overload situation for the 1-node cluster with multiple YCSB clients and adding an additional node to the cluster at runtime, measuring the throughput progress during all steps. An overload situation is reached as soon as the throughput drops with an increasing number of client. The overload situation is experimentally induced for each database.

As Figure 23 depicts for both benchmarks setups, all YCSB client VMs and database nodes (VMs) are located in the OpenStack cloud of UULM to simulate the typical cloud context. It is ensured that YCSB clients and database nodes do not rely on the same physical machine. Further dedicated physical servers are selected to perform the benchmarks in order to reduce the risk of whether other cloud services will affect the results. All benchmarks are run multiple times to ensure stable results. Each database VM is configured with 4 cores, 8GB of memory and 80GB of disk as such a configuration is recommended by the Couchbase and Apache Cassandra

documentations^{61,62}. Each YCSB client is configured with 4 cores, 2 GB of memory and 10GB of disk as the YCSB mainly consumes CPU to generate the load.

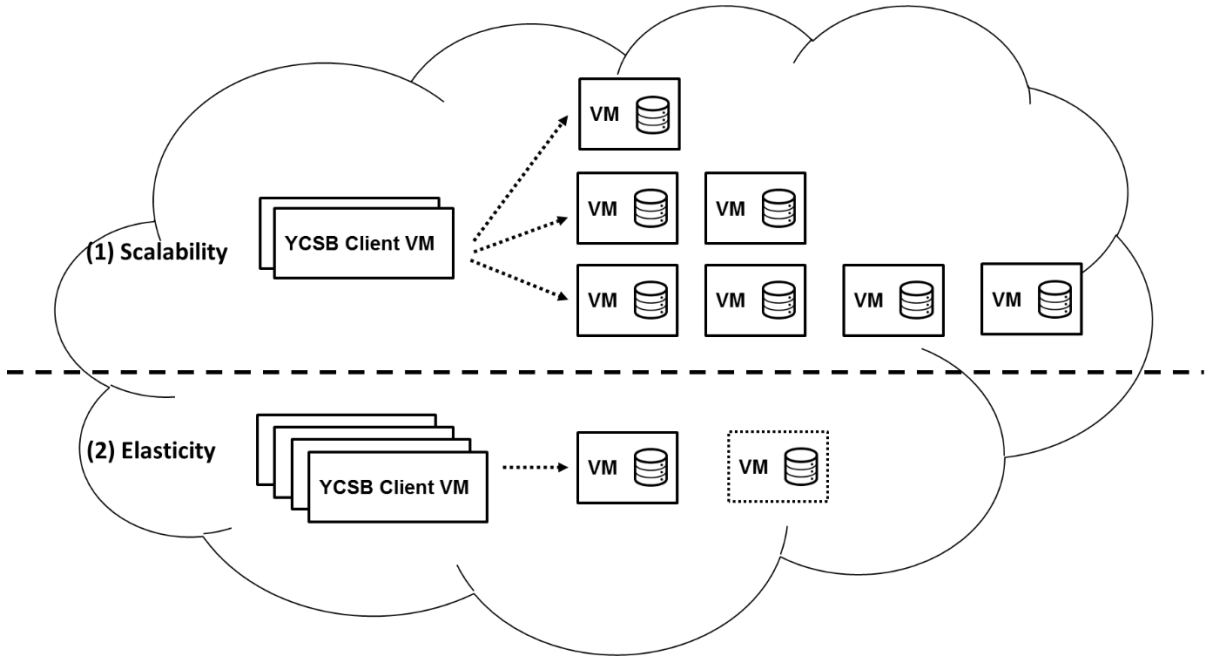


Figure 23 - Benchmarking setups

The scalability benchmark setup encompasses three different workloads: *create-only*, *read-update* and *read-heavy*. The specific CRUD ratio for each workload is shown in Table 5. The *create-only* inserts 1.000.000 records in the database for *read-update* and *read-heavy* workloads, which execute 10.000.000 operations on the records. All workloads use the Zipfian distribution for load generation [95]. The elasticity benchmark is performed with the *read-heavy* workload. All databases are configured to use 6GB of memory and the lowest replication degree. In order to have a comparable setup no further database specific configurations are applied.

Workload	create-only	read-update	read-heavy
Create	100	0	0
Read	0	50	95
Update	0	50	5
Delete	0	0	0

Table 5 - YCSB workloads CRUD ration in %

⁶¹ <http://docs.couchbase.com/admin/admin/Install/install-resourceReqs.html>

⁶² <https://wiki.apache.org/cassandra/CassandraHardware>

4.2.2.4 Results

In the following, the results for the scalability and elasticity benchmarks are presented. The scalability results are presented as a comparison table of the three databases for each workload. The elasticity benchmarks are presented as time series for each database separately and finally compared in the conclusion.

4.2.2.4.1 Scalability Results

Table 6 shows the throughput results for the *create-only* workload as a comparison between the three databases. For each database, Table 6 contains the three different database cluster configurations. Regarding the overall throughput only none of the databases increases the throughput significantly with an increasing cluster. Moreover, MongoDB even decrease their throughput in a 2- and 4-node cluster. Regarding the scalability, none of the databases scales with a growing amount of nodes for the *create-only* workload.

Cluster Configuration		Apache Cassandra	Couchbase	MongoDB
1 - Node	Avg. Throughput (ops/s)	21800	21700	26100
2 - Nodes	Avg. Throughput (ops/s)	17200	24400	13400
4 - Nodes	Avg. Throughput (ops/s)	17000	22300	14700

Table 6 - create-only workload results

Table 7 presents the results for the *read-update* workload. Whereas MongoDB and Apache Cassandra achieve a similar throughput in a 1-Node setup, Couchbase achieves a significant higher throughput (across all configurations). Regarding the scalability Apache Cassandra achieves an 11% throughput improvement from the 1- to the 2-node cluster and a 30% improvement from the 2- to the 4-node cluster. Couchbase improves its throughput from 1- to 2-node cluster of 10% and to the 4-node cluster again of 14%. Scaling MongoDB does not improve the throughput, it even decreases the throughput.

Cluster Configuration		Apache Cassandra	Couchbase	MongoDB
1 - Node	Avg. Throughput (ops/s)	14400	41200	16400
2 - Nodes	Avg. Throughput (ops/s)	16100	45700	13900
4 - Nodes	Avg. Throughput (ops/s)	23000	52600	12800

Table 7 - read-update workload results

Cluster Configuration		Apache Cassandra	Couchbase	MongoDB
1 - Node	Avg. Throughput (ops/s)	15500	45800	32000
2 - Nodes	Avg. Throughput (ops/s)	15500	50000	14300
4 - Nodes	Avg. Throughput (ops/s)	21000	54900	14000

Table 8 - read-heavy workload results

Table 8 shows the results for the *read-heavy* workload. As in the previous benchmark results MongoDB achieves a lower throughput than Apache Cassandra and Couchbase. For the scalability, Apache Cassandra increases its throughput with the 4-node cluster of 27%. Couchbase achieves a throughput improvement from the 1- to the 2-node cluster of 9% and from the 2- to the 4-node cluster again 9%. Couchbase improves its throughput from 1- to 2-node cluster of 9% and to the 4-node cluster again of 9%. However scaling MongoDB up to 4 nodes does not improve the throughput, it even decreases the throughput.

The benchmark results show that there are significant performance (throughput) differences in general between different types of NoSQL databases, where MongoDB achieves the lowest throughput for all workloads, Apache Cassandra and Couchbase achieve similar throughput for the create-only benchmark and Couchbase achieves the highest throughputs for the *read-update* and *ready-heavy* workloads. Regarding the scalability, MongoDB does not benefit from scaling its cluster. It even decreases the throughput. For the create-only workloads Apache Cassandra and Couchbase only achieve a slight throughput improvement. For the other workloads Apache Cassandra and Couchbase, achieve a throughput increase by adding more nodes to the cluster, where Apache Cassandra reaches the highest rate of increase.

For all results, it is taken into consideration, that Apache Cassandra and MongoDB do not allow a configuration with no replication factor, whereas Couchbase allows running without replication. This leads to a slightly better benchmark starting position for Couchbase.

4.2.2.4.2 Elasticity Results

As introduced in section 4.2.2.3 the elasticity benchmark will produce an overload situation for the respective database and will add a node to the cluster while the load is continuing. For each database, the overload situation is determined individually to determine the required amount of YCSB clients to produce an overload situation.

Figure 24 presents the resulting time series of the elasticity benchmark for Apache Cassandra. The overload situation for the 1-node cluster is reached after approx. 40s by starting up to four YCSB clients running load on the Apache Cassandra node. The operations per second start to decrease and after approx. 100s a new node is added to the Apache Cassandra cluster. Internally Apache Cassandra starts to redistribute the data across the two nodes while the YCSB clients still produce load. As the throughput stabilises at 200s as time series indicates, the

redistribution is finished. This result shows that adding nodes to Apache Cassandra at runtime overcomes the overload situation. However, the 2-node cluster does not achieve the peak of ~ 16000 operations per second.

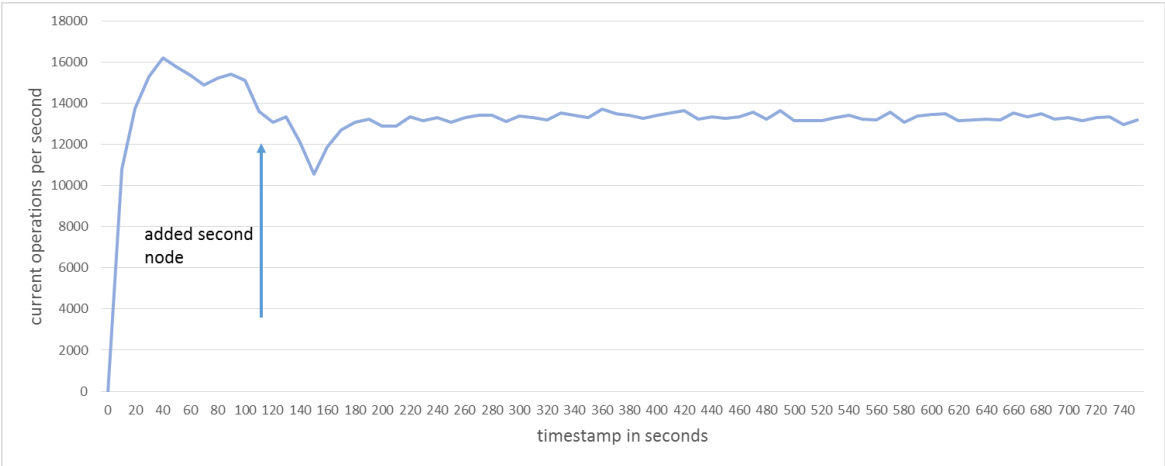


Figure 24 - Apache Cassandra Elasticity Benchmark

Figure 25 shows the time series of the Couchbase elasticity benchmark. Again, multiple YCSB clients overload a 1-node cluster. Similar to Apache Cassandra, it requires also four YCSB clients to overload Couchbase. As Figure 25 depicts is the overload situation reached after approx. 60s. A second node is added to the Couchbase cluster and Couchbase internally redistributes the data. The second node is sufficient to handle the overload situation and as the time series shows the redistribution is finished at approx. 90s. The resulting time series also shows that the throughput has increased with the 2-node cluster compared to the starting 1-node cluster.

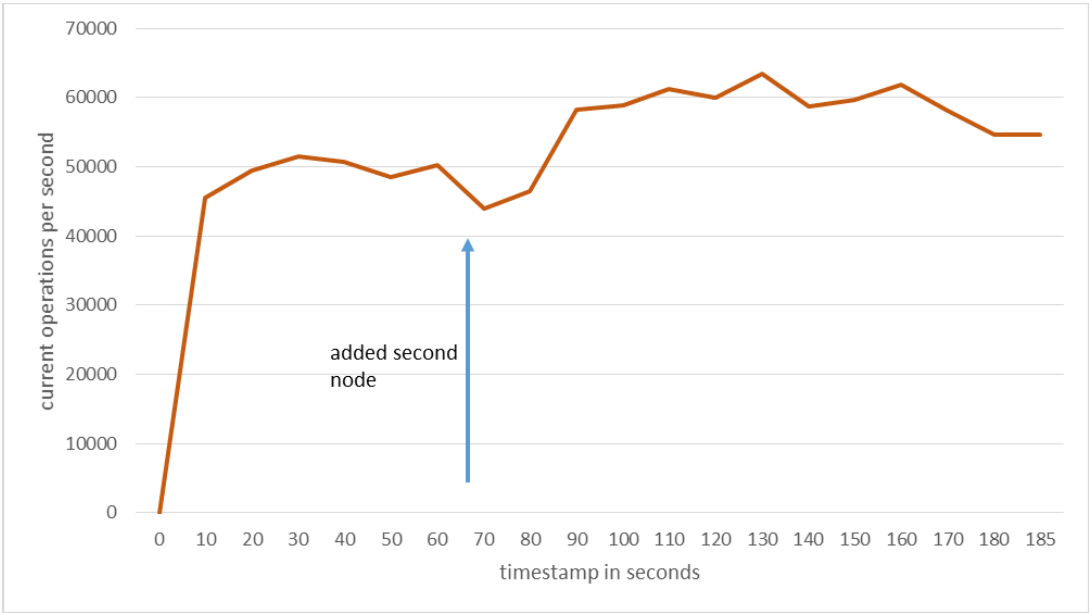


Figure 25 - Couchbase Elasticity Benchmark

Figure 26 shows the time series of the MongoDB elasticity benchmark with multiple YCSB clients overload a 1-node cluster. As Figure 26 depicts the overload situation is reached after approx. 60s and a second node is added. The time series shows that the overload situation cannot be overcome as the throughput drops multiple times significant

and a 2-node cluster does not increase the throughput. Regarding the scalability results of MongoDB, this behaviour could already be expected.

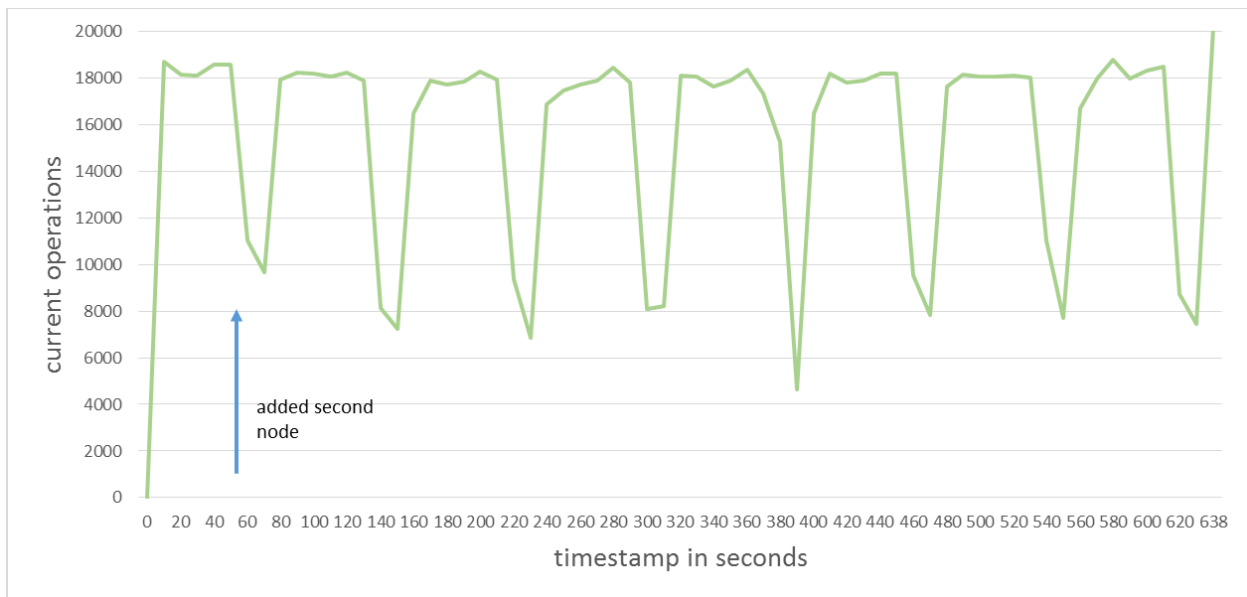


Figure 26 - MongoDB Elasticity Benchmark

4.2.2.4.3 Conclusion

The scalability results of 4.2.2.4.1 and elasticity results of section 4.2.2.4.2 have shown that there are significant differences between the evaluated databases scalability and elasticity. In general, Apache Cassandra and Couchbase achieved a clearly higher throughput than MongoDB in all evaluated scenarios. Regarding the scalability Apache Cassandra increases its throughput with larger clusters. However, the elasticity benchmark has shown that extending the Apache Cassandra cluster under load will solve the overload situation but the throughput increase is not significant. Couchbase also achieves scalability with a growing cluster size but the throughput increase percentage is not as high as with Apache Cassandra. Regarding elasticity, Couchbase shows the best results of the three evaluated databases by overcoming the overload situation and increasing the throughput under ongoing load.

These results show that the selection of a scalable and elastic storage backend for monitoring engine is not a trivial decision due to the varying scalability and elasticity capabilities. As the results show, Apache Cassandra or Couchbase might be a first appropriate solution to provide scalability and elasticity to the monitoring engine. This gained knowledge might also be further used to enrich the semantic description of services relying on one of these databases, e.g., defining a scalability and elasticity grade on the service description level.

4.2.2.5 Future Evaluation Scenarios

With the gained knowledge of the scalability and elasticity benchmarks in a rather small and artificial setup, a starting point for more complex and use-case centered benchmarks is provided. The evaluation of a larger scale distributed database cluster (>20 nodes) constitutes one aspect of further benchmarks. In addition, more monitoring related operations, i.e., aggregation operations, will be included in the benchmarks. With a further extension of the YCSB it will also be possible to benchmark specific time-series databases (T SDBs) which are typically built on top of NoSQL databases.

4.2.3 UULM Approach

As explained in section 4.1, the vendor lock-in not only affects the deployment of applications in the cloud, it also affects the monitoring of applications across multiple cloud providers and across different service levels. This leads

to the challenge of supporting not only multi-cloud but also cross-cloud monitoring. For pure multi-cloud systems, the monitoring tools of the currently selected cloud operator can gather the monitoring data. While basic monitoring data may come free on some cloud providers, often more advanced metrics either cost (Amazon, Rackspace) or require the operator to set up additional monitoring tools. For cross-cloud monitoring, using the providers' monitoring infrastructure is technically feasible, but it increases tremendously the effort, as multiple tools have to be used in parallel. Moreover, it is difficult to assess metrics that involve the crossing of provider domains (such as network traffic from provider A to provider B). Furthermore, it is hard to assess application-specific metrics. In addition, a sophisticated and configurable aggregation on the metrics is currently not easily possible.

The monitoring approach followed by UULM provides a generic and extensible monitoring engine, offering the capability to reduce the cross-cloud provider network traffic and hence reduce costs, enabling a powerful API to customize the monitoring at run-time and a self-scalable architecture [100]. The monitoring system is part of the Cloudiator framework. The following sections describe the UULM approach with its main components and features.

4.2.3.1 Monitoring Agent: Visor

In order to be able to gather the raw monitoring data on the IaaS level from the VMs and component instances, Visor⁶³ is introduced as a monitoring agent. Visor is deployed on every VM orchestrated by Cloudiator and provides a remote interface in order to configure a particular Visor instance at deploy and at run-time. Figure 27 depicts Visor's main functionality. The dynamic configuration of Visor allows the close mapping to the application by also only collecting the required metrics, thus saving space and bandwidth. Visor supports the capturing of data on a per component instance basis as well as on a per-VM basis. The former is achieved by sensors monitoring basic system properties on virtual machine level, e.g. by accessing system properties such as CPU load. The latter is performed by exploiting the fact that all component instances run inside a Docker container (cf. section 4.1.2.1) and the resource consumption can be retrieved on a per-container basis. By default, Visor offers various sensors supporting system metrics such as CPU load, memory consumption, disk I/O, and network I/O. In order to support custom metrics, like the request rate of a web server, Visor supports the implementation of custom sensors, by providing an easy-to-implement Java interface. It exploits the dynamic class loading properties of Java in order to be able to add those implementations at runtime.

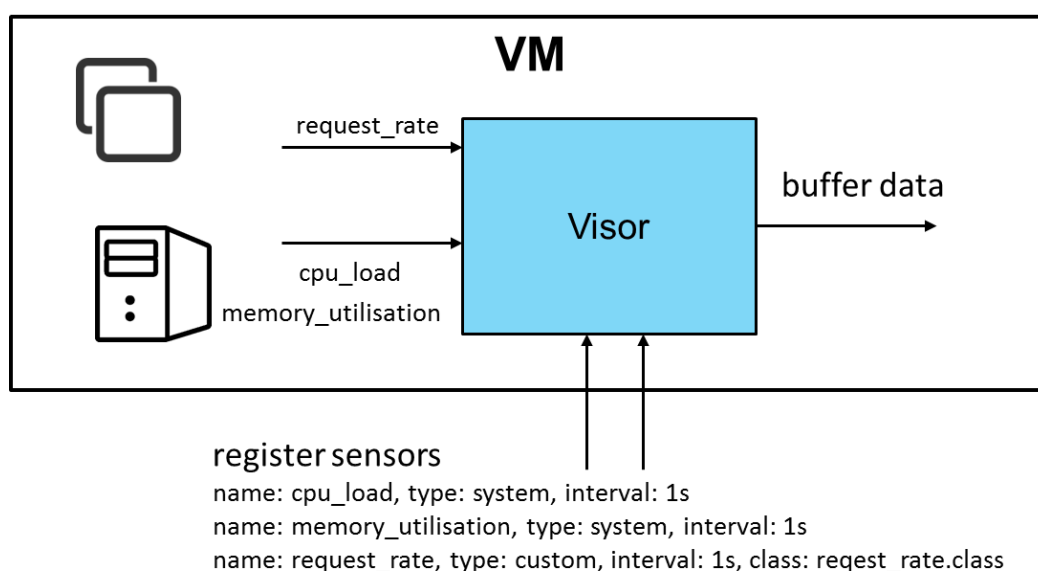


Figure 27 - Visor

⁶³ <https://github.com/cloudiator/visor>

4.2.3.2 Aggregation Levels

In order to provide the various raw metrics gathered by Visor to further consuming entities like the SLA Engine, the raw metrics need to be aggregated. Aggregation includes for instance the computation of average, minimum, maximum or simply the normalisation of values. In addition to that, aggregation may include merging of metrics, e.g., when computing the average of averages. Hence, aggregation is always BPaaS bundle specific, depending on the specified metric model in the CAMEL file. In order to satisfy the requirement for minimum network traffic and scale of the monitoring system, the aggregation is performed as close to the data source as possible. Table 9 provides an overview of the different aggregation levels with their respective input, aggregation location and output.

Scope	Input	Aggregation location	Output
host	single VM	local VM	local storage
cloud	VMs in cloud A	any VM in cloud A	shared storage (inside cloud A)
global (cross-cloud)	VMs from at least two clouds	home domain	storage at home domain

Table 9 - Aggregation Levels

All aggregations that require input data from a single VM will be performed on this VM. We refer to this computation to happen in the host scope. For this approach, only a local storage is accessed and no communication is required which further reduces latency. Aggregations that need input only from VMs from a particular cloud are performed in the cloud scope. Such computations exclusively access the shared space (shared T SDB in Figure 28) spanning a cloud. While it is desirable to distribute all computations of a particular cloud scope amongst the affected VMs, the definition of a suitable algorithm is currently work in progress. Finally, computations that require input from multiple clouds happen in a cross-cloud scope (or global scope). These are performed in the home domain of Cloudiator.

4.2.3.3 Distributed Architecture

Figure 28 provides an overview of the general distributed monitoring architecture with a sample application consisting of two VMs at cloud provider A (Amazon) and another VM at cloud provider B (Openstack). Each VM contains a Visor and Aggregator⁶⁴ instance besides the actual application components; the respective aggregation level explicated in Table 9 is indicated by the colouring scheme.

A key element when computing higher-level metrics especially over larger time-windows is the need to buffer raw monitoring data. T SDBs have been designed to store timestamped data in an efficient way and also to provide quick access to the stored data. Many T SDB implementations support applying functions on stored data right out of the box what makes them a perfect match not only for buffering, but also for aggregation [83]. The T SDB approach needs to be able to work with limited resources to not limit the actual application and increase available resources when more VMs are being used. In order to cope with these requirements, the following approach is followed: from each VM acquired for an application, we reserve a configurable amount of memory and storage (e.g. 10%) that we further split between a local storage area and a shared storage area. Both storage areas are managed by a T SDB instance running on the VM. The Visor instance running on this VM will then feed all monitoring data to the T SDB. The T SDB will store data from its local Visor in the local storage area and further relay the data to other T SDBs where such data is stored in the shared storage area. This feature avoids that a T SDB becomes a single point of failure, but still enables quick access to local data. In order to keep network traffic between cloud providers low, any T SDB will only select other T SDBs running in the same cloud to replicate its data. Hence, this concludes

⁶⁴ <https://github.com/cloudiator/axe-aggregator>

to a ring-like topology that has been introduced in peer-to-peer systems [101] and is also used by distributed databases [94].

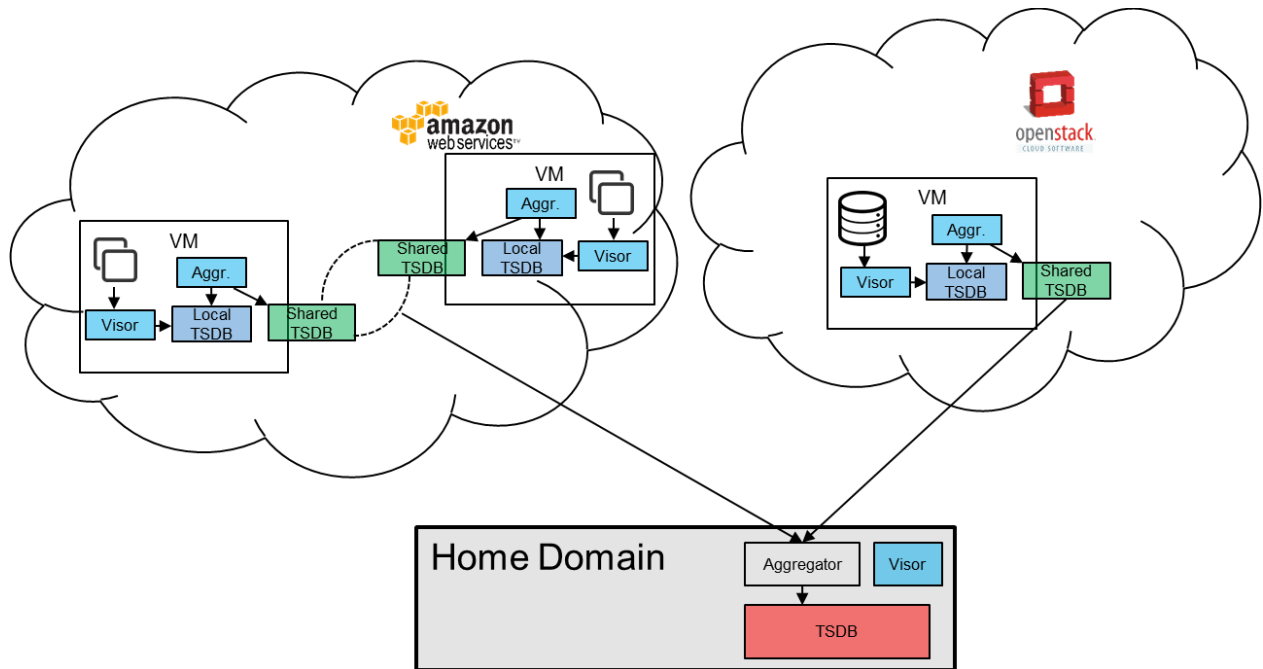


Figure 28 - Distributed Monitoring Architecture

This approach leads to an automated scaling of the monitoring infrastructure with the increasing amount of VMs as the size of the TSDB cluster in each cloud grows with the amount of VMs. Therefore, the TSDB provides for scalability and elasticity. As a first approach, KairosDB65 with Apache Cassandra as storage backend is employed as the TSDB for all domains.

4.2.3.4 Generic TSDB Layer

As the benchmarking results of section 4.2.2 have shown there are significant differences in the scalability and elasticity of the analyzed distributed databases. As the current monitoring architecture is currently bound to one specific TSDB, KairosDB, a more generic TSDB integration will be beneficial for the distributed monitoring architecture. Abstracting specific TSDBs through a generic TSDB API will allow the usage of different TSDBs for the respective aggregation level and the resulting requirements. Whereas on the host aggregation level a low resource consumption is desirable (preferable in-memory operation), on the cloud aggregation level scalability and elasticity are added to the requirements. For the global aggregation level, more complex aggregation functionalities on the TSDB side are beneficial to support the aggregation component in the home domain.

Currently the design and implementation of such a generic *TSDB abstraction layer* is an ongoing process at UULM with the focus on the scalability/elasticity capabilities of the TSDB storage backend and the actual aggregation capabilities of the TSDB. Regarding the scalability/elasticity capabilities, the results of section 4.2.2 provide first basic knowledge, which will be extended to more specific time series related benchmarks (cf. section 4.2.2.4). Regarding the aggregation capabilities, an analysis of common TSDBs (cf. Table 10) has shown that basic statistical operations like *min*, *max*, *average*, *median*, *sum* are widely supported by the TSDBs considered; however the support of more complex operations and the combination of multiple operations depends on the specific TSDB. Further, the support of the automated reduction of outdated monitoring data and continuously running operations

⁶⁵ <https://kairosdb.github.io/>

is only supported by a subset of TSDBs. Table 10 shows an architectural and feature comparison of four common TSDBs.

Name	KairosDB		OpenTSDB	InfluxDB	Prometheus
Version	1.0.0		2.1.0	0.13	0.19.1
Datastore	H2	Apache Cassandra	Apache HBase	Proprietary (LSM Tree based)	Proprietary (file based)
Distributed	No	Yes	Yes	Yes	No
Replication	No	Yes	Yes	Yes	No
In-memory	Yes	No	No	No	Yes
Reduction	No		No	Yes	Yes
Unique features				continuous operations, combination of operations	continuous operations, rule processing

Table 10 - TSDB feature comparison

The first version of the TSDB abstraction layer will bring together the specific aggregation operations of the analysed TSDBs in one API. This API will be built in a modular way to provide an easy integration of further TSDBs. In addition, the API will also integrate common NoSQL databases like Couchbase and use them as a TSDB. As common NoSQL databases do not offer aggregation capabilities in the extent of TSDBs, the missing aggregation capabilities have to be implemented in the abstraction layer.

4.2.4 FORTH Approach

FORTH has developed a distributed cross-layer monitoring framework [102] which is part of its overall cross-layer adaptation framework. In the context of this project, this framework is updated while a particular cross-layer quality model has been devised. The latter quality model can be used for selecting those metrics that can be exploited to form user/BPaaS requirements as well as for explicating the way metrics can be computed in the same as well as across different layers. Both contributions are now shortly analysed in the following two sub-sections.

4.2.4.1 Distributed Cross-Layer Monitoring Framework

The cross-layer monitoring framework of FORTH was designed mainly to cover all layers and be able to exploit cross-layer quality models. Its main idea was that measurements at different layers are encapsulated by sensors attached to respective layer-specific components and that these measurements are stored in a highly efficient complex event processing engine like Esper. Then, this event processing engine could take care of aggregations and enforcing the respective cross-layer dependencies. As measurements directly map to events if their value is compared to conditions, the event processing engine was also employed in order to not only report simple but also complex event patterns that could be used to trigger corresponding adaptation rules. For this kind of reporting, a publish-subscribe mechanism is used to also enable the distribution of the adaptation functionality enabling different instances of an adaptation engine to subscribe to different partitions of events. For example, a particular instance of an adaptation engine could focus only on rules involving events covering the security aspect.

While the original version of the framework was able to enforce measurability via the respective cross-layer quality model exploited along with the respective sensing mechanisms deployed, its architecture was considered simple and not fault-tolerant in the sense that the event processing engine constituted a single-point-of-failure. In addition, it did not focus on addressing layer-specific scalability aspects, thus actually prescribing only a high-level coarse-grained monitoring architecture. To this end, in the context of this project, this architecture has been refined and such a refinement now guides the update to the development of the respective research monitoring prototype. This refinement exactly attempts to address the shortcomings of the initial architecture.

First, the refined architecture does not involve a single point of failure. This is enabled by replicating components as needed and where possible. Second, the architecture now considers layer-specific scalability aspects by attempting to scale the monitoring system when needed as well as to replicate the information stored in order to be more fault-tolerant. Third, the event production has been decoupled from the measurement aggregation while measurement aggregation has become more focused by being applied only on the layer it maps to.

An overview of the architecture is provided in Figure 29 - The logical architecture of FORTH's monitoring framework. As it can be seen, the architecture is split into 5 main parts: (1) an event production and publishing part; (2-4) three layer-specific parts focusing on the sensing and aggregation at the same layer; (5) a cross-layer dependency part facilitating the propagation of dependencies on the different layers. The event production and publishing part retrieves the measurements from each layer via a publish-subscribe mechanism, assesses the respective conditions and produces events that are stored in an event database which is replicated/backed-up. This part relies on an event processing engine to produce complex event patterns. It also enables the retrieval of all types of events produced via a publish-subscribe mechanism.

The layer-specific parts have the freedom to exploit any kind of measurement database that can assist in the respective storage and measurement aggregation. As such, the UULM effort over providing an API, which integrates the functionality of different TSDBs, could be quite advantageous here (See section 4.2.3.4). Thus, both TSDBs or complex event processing engines could be used or any other kind of suitable database. Apart from the measurement database itself, each layer employs respective sensors as well as an aggregation component that is able to aggregate the information produced by the sensors and being stored in the database. The implementation of this component depends on the level of automation that exists in the respective database. In the case of a complex event processing engine, the role of this component is limited. It could take the responsibility to transform sensor measurements to events as well as to initially produce and load the aggregation rules into the engine. Thus, aggregation is more or less performed automatically by that engine. In case of a TSDB, this depends on the level of automation offered by the respective implementation (See section 4.2.3.4). In Kairos TSDB, for example, only measurements can be stored and thus the aggregation functionality has to be totally performed by the aggregator. In other TSDBs like InfluxDB, some aggregation mechanisms are in place so the aggregation could be more or less automated with few exceptions. In any case, we consider that in some cases, the aggregation functionality can be limited with respect to the aggregation functions that are available. In this sense, the aggregator will act as a complementary counterpart that offers the missing aggregation functionality and has the responsibility to realise the respective aggregations that have to be performed. Finally, we should note that each layer-specific part offers a publish-subscribe mechanism in order to propagate information to interested subscribers which can take the form of the event generation and publishing part and the cross-layer dependency part.

The cross-layer dependency part has the responsibility to subscribe to measurements of one or metrics and propagate them to the respective layer-aspect part by considering the QoS dependencies of the cross-layer quality model. The propagation maps just to storing the respective measurement on the corresponding layer's measurement database.

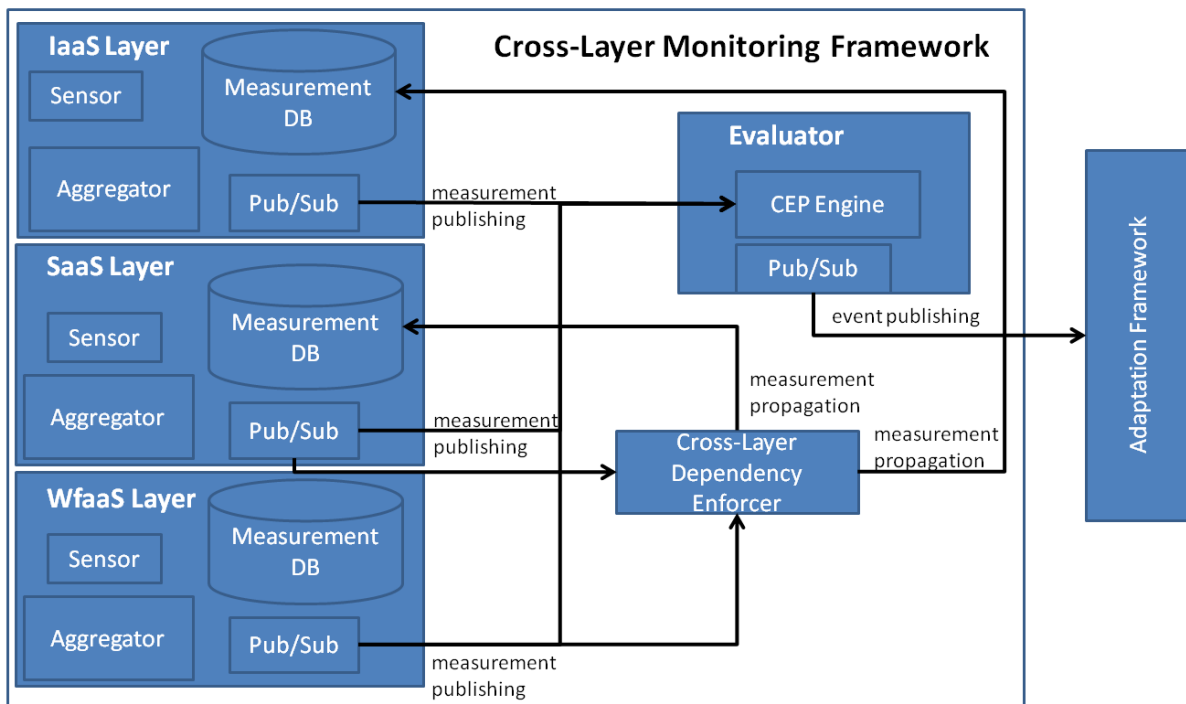


Figure 29 - The logical architecture of FORTH's monitoring framework

On the physical level, the mapping of logical components to respective instances depends on the layer involved and to the nature of the components. On the IaaS layer, we take the view that monitoring is performed in an intrusive manner by placing sensors over the user VMs in order to measure low-level metrics (similar to UULM approach - see section 4.2.3). These sensors then report measurements on measurement databases coupled with respective aggregators. The latter couple of components could be mapped to the physical level in two alternative ways. We could have 2 (or even more) measurement databases and respective aggregators in so called Management VMs that are placed in each cloud. In this sense, the aggregation overhead is split between the aggregation components and we enable a respective degree of replication between the measurement databases. This mapping leads to high communication needs as each sensor is configured to report all measurements in a respective measurement database. Another mapping approach is to have a more distributed architecture where the triangle of sensor, measurement database and aggregator is deployed on each user/BPaaS VM. This of course creates some overhead, which should not be significant, to the corresponding VM but has the main advantage that the main sensing and aggregation logic pertaining to a specific VM stays mainly on that VM and is not moved to other triangle placements. Each triangle in a cloud can replicate some information on other triangles. However, this information could be only specific to our analysis and aggregation needs, which can mean replicating only meaningful measurements like aggregations over CPU metrics. This alternative mapping reduces the communication overhead and can be controlled to exhibit different replication levels. Therefore, it seems to be more suitable than the first one. In order to cater for the aggregation at a global level in the IaaS layer (e.g., to calculate average CPU in one cloud or all clouds exploited by a BPaaS), enabling the computation of composite IaaS metrics at the same or across cloud, the database-aggregator couple needs to be deployed in 2 VMs, one constituting the centralised aggregation couple for the layer and the other its back-up. To this end, we enable the calculation of the composite metrics via the replication of the relevant information produced in each cloud by storing it in the centralised level. As such, we also enable to have a stable publish/subscribe mechanism, which can be offered to interested components.

Concerning the SaaS layer, we actually have two main types of services, which can lead to different physical deployment options. External services are out of control of the system. In this respect, they can only be measured

Copyright © 2016 UULM and other members of the CloudSocket Consortium
www.cloudsocket.eu

when the Workflow Engine performs the respective calls. As such, it seems that their measurement could be handled at the same place where a Workflow Engine is hosted. On the other hand, an internal service component is placed on user/BPaaS VMs. In this respect, it coincides with such VMs and could be handled in a similar way. In this respect, our proposal includes the following. Internal service measurements are handled by measurement database of the user/BPaaS VMs and are produced by sensors, which are placed on those VMs. Thus, respective aggregations can take place inside the same measurement database with respect to IaaS metrics. As such, we have one measurement database, different sensors (as they capture different type of information) and logically speaking different aggregators to split the aggregation functionality and exploit the advantages that multi-threading provides. Concerning external service measurements, these can be handled by sensors that are attached or placed in the VMs hosting the Workflow Engine(s), which execute the BPaaS workflows. As we will see later on, in these VMs, a measurement database for workflow metrics as well as respective aggregator will be involved. Thus, similarly to the case of the internal service measurement, the measurement database will be in common but the sensing and aggregation functionality will be split.

To handle again the global level at SaaS layer (e.g., to calculate mean response time of a service over all BPaaS workflow executions), we expect that again we need 2 VMs. These VMs could be the same as those for the IaaS global level thus leading to the sharing of the measurement database and the split of aggregation functionality. The same can hold for the global level at the WaaS layer only in case workflows and tasks are shared between many workflow engines and not just one.

Concerning the WaaS layer, things seem to be similar. We deploy a triangle in each VM hosting a workflow engine. Replication of information between engines can take place to cater for the appropriate back-up/replication of the information stored. The overall physical deployment architecture can be seen in Figure 30. As it can be seen, there are as many VMs as the number of user/BPaaS VMs and VMs hosting the workflow engines plus VMs catering for the global and the event publishing levels (along with their respective back-up for the latter). As such, the extra cost of monitoring is small, as we actually need only four additional VMs to cover the global level and the event publishing one. In case the load at the global level is big, leading to a reduction of the respective aggregation or evaluation performance, then new VMs could be deployed on demand with which a split of the respective functionality could be achieved to better load balance the monitoring framework. Moreover, we need to stress that the architecture is quite fault-tolerant in the sense that the global and reporting/evaluation level is backed-up. The failure of a user/BPaaS or Workflow Engine VM will usually lead either to its re-start or the generation of a new VM by the adaptation BPaaS system/framework while the measurement data will not be lost as they will also be replicated on other VMs.

4.2.4.2 Cross-Layer Quality Model

A quality model is a specific of a set of quality terms along with their relationships. Such quality terms span quality groups, attributes and metrics. Groups (e.g., performance) enable a specific partitioning of the term space, while metrics (e.g., average response time) provide the necessary details in order to measure specific attributes (e.g., response time) of components (e.g., BPaaS, workflow, task, etc.). In this sense, a group encompasses various terms, while an attribute can be measured by one or more metrics. Moreover, attributes can be composite or simple mapping to more abstract or concrete properties. Attributes can also be measurable or not. In addition, metrics can be raw or composite. Raw metrics (e.g., raw response time) can be directly measured from sensors or the component's instrumentation system. On the other hand, composite metrics (e.g., mean response time) can be computed by applying specific formulas (e.g., mean) over metrics, attributes or constants.

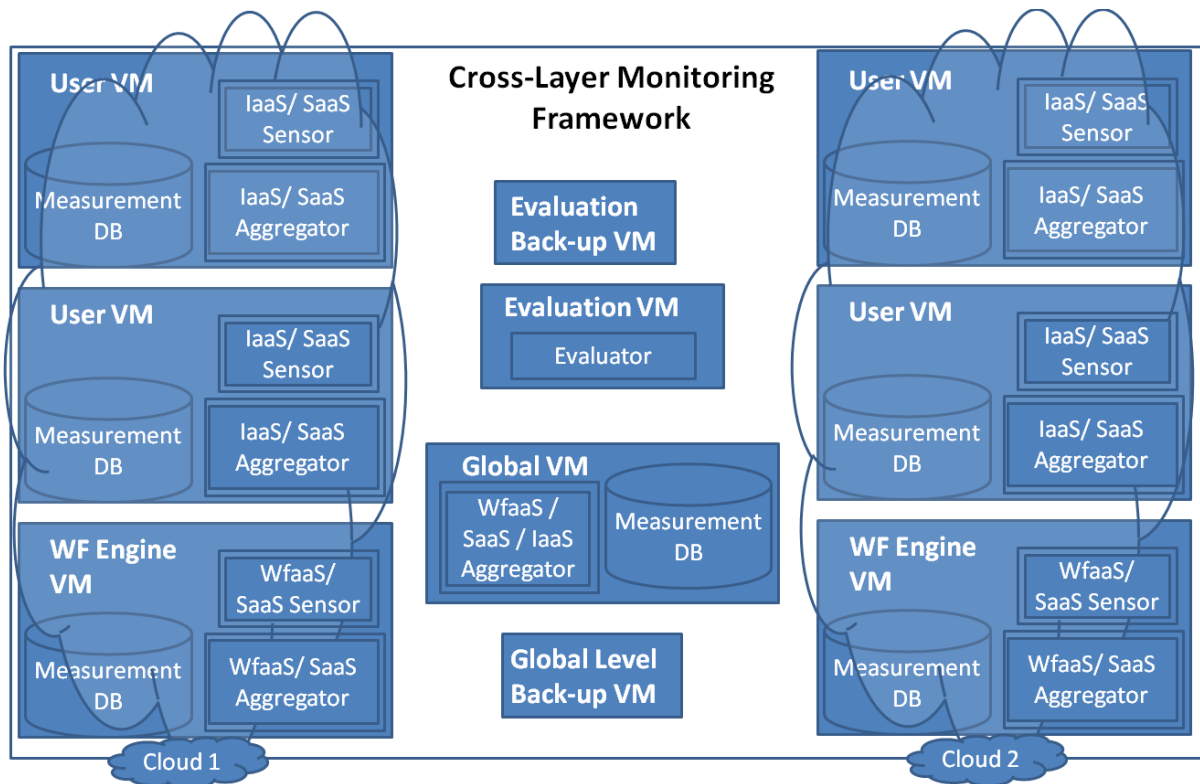


Figure 30 - Physical architecture of the FORTH's cross-layer monitoring framework

Based on the above analysis, it is apparent that quality models actually explicate how measurements can be performed either at the same or across different levels as they explicate how values obtained from sensors can be aggregated in order to produce measurements in respective higher-levels. Quality models can be considered to include metric derivation trees, which enable this kind of propagation. As such, quality models are the main instruments to guarantee measurability.

Quality models can be specified at different layers of abstraction. Indeed, we have seen layer-specific quality models proposed for the WfaaS, SaaS, PaaS and IaaS layers. However, as they include relationships and dependencies between quality terms, they can also be used to guarantee cross-layer measurability by connecting quality metrics defined at different abstraction layers. As such, cross-layer along with layer-specific quality models can lead to the production of a global quality model that can guarantee the measurability across all the layers that are relevant in the context of BPaaS services.

Such a global quality model has been recently proposed by FORTH in the context of this project. Its overview is depicted in Figure 31. This model covers three main layers, WfaaS, SaaS and IaaS, and includes a limited number of dependencies among these layers, where some dependencies apply between WfaaS and SaaS and others between SaaS and IaaS. This model has been derived by considering the literature with respect to different layers as well via the devising of new quality terms to cover aspects not touched or improperly addressed in the literature. In the following, we shortly analyse the content of this quality model for each layer and then we explain in short the nature of the cross-layer dependencies that have been defined.

Concerning the workflow layer, the quality terms have been split according to the quality groups of time, reliability and cost. For each group, the terms defined map to both the workflow and task level. In many cases, metrics at the task level are used to compute similar metrics at the workflow level. In this computation, the structure of the workflow can play a role (this has also been witnessed in service research where service concretisation involves particular aggregation formulas, which explicate the way the performance of the service selected per each task propagates to the performance at the workflow or composite service level).

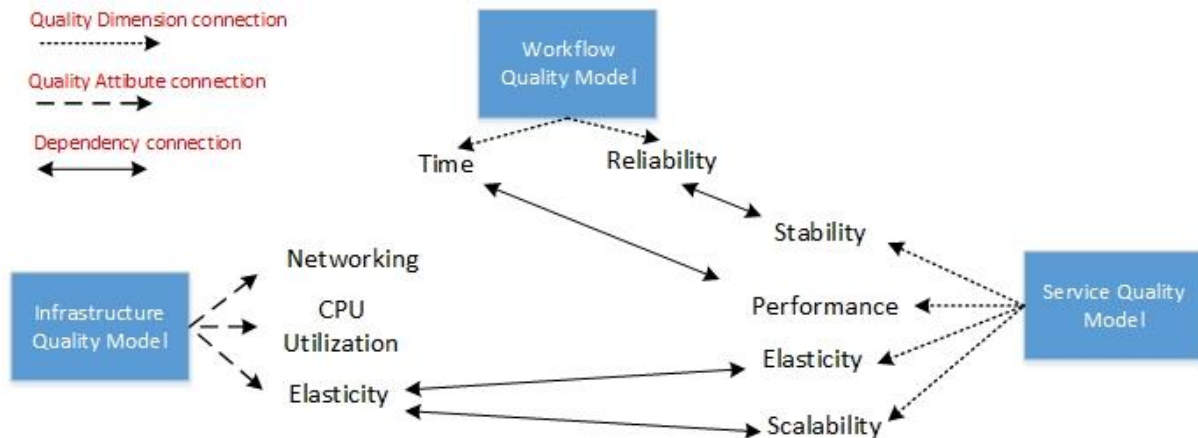


Figure 31 - Overview of the cross-layer quality model

Time-based metrics involve the whole processing time of the workflow which can be split into the total execution time and the total delay time (covering intra-task as well inter-task (transition) delays). Reliability metrics involve attributes mapping to workflow availability, reliability and fidelity [103] as well as respective metrics able to measure them (such as mean time between failures, MTBF). Special attention was put to definition of the fidelity metric that assesses how well a component/service satisfies the requirements posed to it. It has been decided that this metric should be first computed independently between the two main levels, workflow and task, as the amount of requirements posed on the latter level could be less than those on the former one (as focus is usually on overall performance and not individual one). Via this rationale, the workflow fidelity is computed by checking whether a set of measurements mapping to the execution history of a BPaaS workflow satisfy the requirements posed on that workflow. The average measurement degree of satisfaction is then computed to produce the overall fidelity value. A similar procedure is followed for the task level where the focus is now on respective task requirements and measurements only. In the future, fidelity computation formula could be slightly modified to account for the age/time of the measurements. Finally, cost metrics are proposed for both the workflow and task level. For the workflow level, the cost is computed from the cost of all the tasks involved plus the management cost of the workflow. The task cost is then split into cost concerning the services and resources exploited to support the respective task execution.

Concerning the IaaS layer, the respective terms have been grouped into the following 6 groups: networking, utilisation, storage, bandwidth, scalability and elasticity. Networking metrics considered span packet transfer time and mean packet loss frequency while utilisation metrics currently include statistical measures over CPU utilisation for single or multi-core architectures. Utilisation metrics will be expanded towards covering the storage aspect. Storage metrics include speed of read and writing and RAM access time. Bandwidth metrics map to statistical measures of bandwidth like maximum bandwidth. Scalability and elasticity metrics have been mainly drawn from respective literature. Concerning elasticity, the metrics considered are the precision of scaling [69] and the mean-time-to-quality-repair (MTQR) [73]. Scalability metrics on the other hand include [73] scalability range and speed.

The SaaS layer was covered by considering some state-of-the-art models [62], [64], [104]. The following groups are included: (a) performance, (b) stability, (c) scalability and (d) elasticity. Performance attributes include execution time, response time and throughput where also the dependencies between some of them are also outlined. Some of the metrics involved are also mapped to more fine-grained metrics that can be easily computed from sensors. Stability is considered to cover both service ability to provide a certain level constantly as well as a stable interface. It includes attributes like reliability and availability, which are measured by respective metrics like MTBF and raw availability. Scalability maps mainly to the metrics of scaling utilisation and precision while elasticity is associated to metrics of mean-time-taken-to-react (MTTTR) and performance-scale-factor.

The cross-layer dependencies currently considered are of the following nature: (a) similar metrics, which go from the SaaS to the task level in the WfaaS layer. For instance, service response time directly maps to the execution time of a workflow (service) task. As can be easily understood, the metrics are similar and have a more or less one-to-one direct mapping but just concern different layers. It has to be noted, though, that this is an over-simplification by assuming that each service task maps exactly to one service. However, in other case, one task could map to a composition of services. In this respect, we would then have to define a specific computation procedure similar to one proposed to cover the gap between the task and workflow level for similar metrics like execution time. In this respect, the task execution time would equal to the aggregated response time of the service composition which would depend also on the structure of this composition; (b) similar mainly elasticity metrics that go from the IaaS to SaaS layer. Again, we have relied on an over-simplification to cover such dependencies but our main goal was to identify the mapping and not formulate it in a respective computation formula. In this way, the MTTTR could be equal to the scalability speed depending also on the resources needed to be scaled (thus mapping to a one-to-one mapping or a mapping that also depends on the amount of resource to be additionally reserved).

We acknowledge the fact the cross-layer dependency model is minimal. In addition, some aspects have been neglected as well as layers. In this respect, the cross-layer quality model proposed will be expanded and this is indicated in more detail in section 4.2.6.1.

4.2.5 Integration / Synergy of Approaches

The main idea for synergy of the two approaches that have been presented in the previous two sections is that each approach focuses on different layers and then there is a global layer covering the generation of the respective events derived from these measurements. The corresponding architecture of the proposed cross-layer synergic approach is depicted in Figure 32. In this respect and by considering the fact that the IaaS and PaaS layers generate most of the monitoring load, it is advocated that the distributed monitoring architecture of UULM is exploited to perform the monitoring at these layers by also employing a distributed T-SDB which has been proven quite robust in handling the respective huge amount of measurements that have to be stored and aggregated. Distribution in this case is addressed by: (a) employing monitoring nodes on different clouds; (b) employing replication mechanisms inside the architecture to address single point of failure.

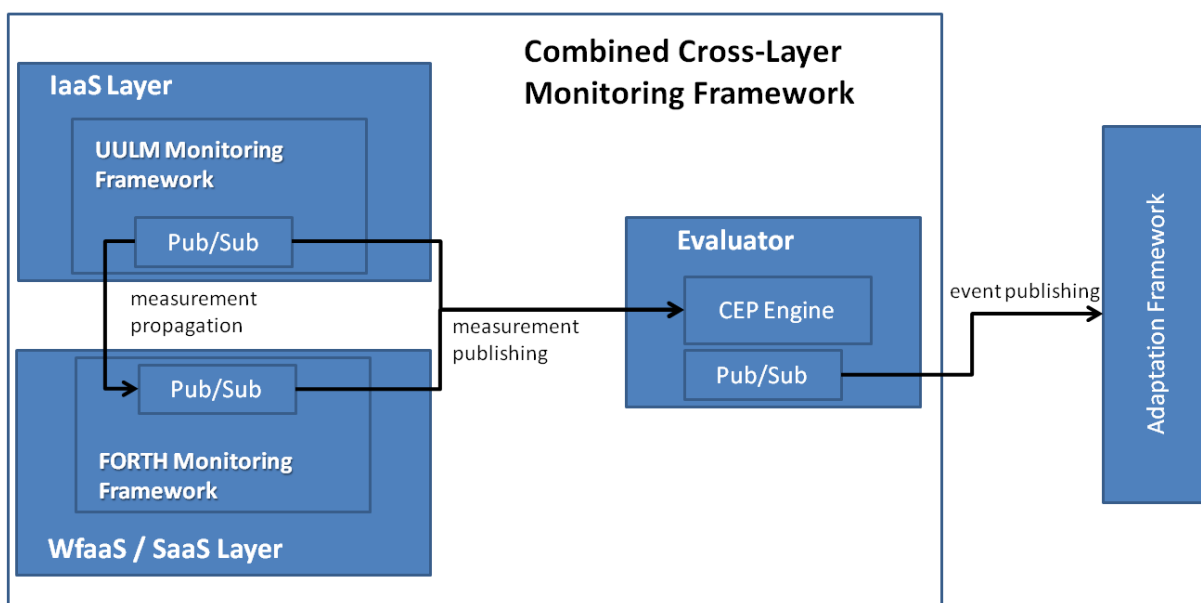


Figure 32 - Combined cross-layer monitoring architecture

On the other hand, by considering that the SaaS and WfaaS layers do not produce a heavy monitoring load and that the respective dependencies are well covered, the distributed monitoring of FORTH can be exploited.

Distribution in this case is addressed by: (a) employing monitoring nodes at the same host or near Workflow Engine instances with the rationale that most of the monitoring is produced by these instances or sensors either attached to them or exploiting log information provided by them; (b) employing similar redundancy mechanisms as in the case of UULM framework application.

The integration itself is confronted via employing publish/subscribe mechanisms between the different frameworks and of course considering the cross-layer metric models which highlight how propagation can be performed across (all) layers. When one measurement at the IaaS or PaaS layer needs to be propagated to the SaaS or WaaS layer, then the monitoring framework of FORTH would have already subscribed to the metric mapping to this measurement. In this sense, it will be able to retrieve it and proceed with the propagation/aggregation of the monitoring information.

All monitoring events that map to SLOs or events in adaptation/scaling rule event patterns are reported on the global layer. This means that the same propagation mechanisms can be employed also in this case. In particular, the event consuming components of the global layer subscribe to metrics, which are involved in the conditions of the events that need to be generated and stored for further analysis and more composite assessment. The event assessment can proceed based on the approach of UULM as reported in section 4.3.2.1. Replication is enforced also for this layer, mainly in terms of the event database being exploited in order to address the single point of failure issues. The global layer is the part of the architecture that communicates with the rest of the components in the Execution Environment. To enable such communication, again the publish-subscribe mechanism is employed with the sole exception that now the reporting/publishing concentrates on events and not measurements of metrics.

4.2.6 Future Research

4.2.6.1 Cross-Layer Quality Model Expansion

The cross-layer quality model (see section 4.2.4) from FORTH still needs some expansion as cross-layer dependencies must be enriched to cover additional metric dependencies in different layers while the PaaS layer must also be covered. In parallel to this expansion, the model must be fully specified in a quality specification language, such as OWL-Q, CAMEL or a combination of these languages. OWL-Q is supported by FORTH's monitoring framework while CAMEL by UULM's monitoring framework. As such, we foresee that OWL-Q is mainly used and then its specifications are transformed, when needed, to produce CAMEL specifications to be exploited by the UULM monitoring framework. Such transformation will be semantics preserving and lossless as OWL-Q and CAMEL are more or less compatible and we foresee including semantic annotations (in OWL-Q) in CAMEL (see section 2.2).

4.2.6.2 Quality Model Realisation

The cross-layer quality model needs to be exploited by the monitoring framework via realising those leaf-level metrics that can guarantee that measurability of the whole quality model hierarchy. For this realisation, sensors should be developed for such metrics and be embedded in the respective monitoring frameworks proposed such that they can then be attached to those components that need to be measured. Such realisation can rely on re-using and possibly extending existing tools and mechanisms.

Some quality dependencies might be BPaaS-specific so they can take a different form depending on the respective BPaaS and the corresponding infrastructure on which this BPaaS is deployed. As such, there should be mechanisms in place to derive such dependencies according to the current BPaaS and its context. Such mechanisms are actually covered by the research direction detailed in section 3.5.3.

4.2.6.3 Synergic Cross-Layer Monitoring Approach

The approach highlighted in section 4.2.5 is still in development. Therefore, it might be modified and adjusted accordingly. In addition, this approach needs to be thoroughly evaluated to check its ability to satisfy respective monitoring requirements covering aspects like measurement and event accuracy, measurement timeliness, as well as the performance and scalability. Such an evaluation could be used as a further feedback to adapt the framework that could also lead to a great degree of state-of-the-art advancement. We believe that this synergic approach will lead to a successful research outcome that could result in common publications and could be adopted by the CloudSocket implementation prototype.

4.2.6.4 Monitoring Adaptation

As indicated in the introduction of section 4.2, any monitoring framework should be robust and flexible to address different situation types, whether foreseen or unexpected. As such, the monitoring frameworks proposed must be able to adapt themselves to address such situations. Such adaptation could involve scaling the framework to address additional load, replacing failing nodes with back-up ones, creating the necessary redundancy or even modifying the measurement frequency, if this is allowed based on the requirements posed, to be able to address the increased monitoring load. Based on the above, research has to be performed resulting in appropriate architectures and methods which appropriately organise the monitoring framework, explicate the way re-organisation can be performed and include smart structures and redundancy mechanisms to guarantee system robustness. For instance, service-oriented architectures could be exploited along with respective adaptation mechanisms proposed for service-based applications and systems to enable the monitoring frameworks to become self-adaptive and robust.

4.3 Adaptation

The capability of a service-based system to adapt itself when critical situations occur is of crucial importance and has a major effect on the gains of a service provider as well as its reputation. For instance, if a service provider provides an unreliable service that constantly violates its SLAs, then it is quite possible that the gains of this provider will be reduced due to SLA penalties as well as due to a reduction in its market share because of customer dissatisfaction and reputation decrease. Maintaining an SLA is not an easy business especially if a service is offered in dynamic environments. This requires sophisticated monitoring and adaptation mechanisms that are able to even detect when a problem (i.e., a SLO violation) will happen and then perform respective actions to prevent it. By focusing on adaptation and the context of BPaaS, it is apparent that in the offering of a BPaaS many different layers are involved including different types of BPaaS components that could fail in a functional or non-functional manner. Such a failure can sometimes occur either concurrently or an ordered manner even across different layers. In this respect, even if layer-specific mechanisms are in place to handle a specific failure or fault, if these mechanisms are not coordinated in a cross-layer manner, then the desired adaptation effects will not be achieved. On the contrary, it is highly probable that either similar actions are used to alleviate the problem or event conflicting ones, where one action diminishes the effect of the previous one. To this end, there is a need to develop cross-layer adaptation systems that can coordinate the layer-specific logic in a suitable manner. By focusing on this issue, we first analyse the state-of-the-art for specific layers as well as for cross-layer adaptation and then we analyse the respective research approaches that have been proposed by two main research partners in the project, UULM and FORTH. As each approach seems to have a different focus, a synergy of the approaches is also proposed which will certainly lead to joint research results. Finally, we conclude with interesting research directions that might be followed in the next project period that can possibly lead to research results, which are incarnated in the next version of this deliverable.

4.3.1 State of the Art

4.3.1.1 Service Monitoring & Adaptation

The need for monitoring different functional and non-functional requirements, as well as for taking adaptation actions is widely recognized by industry and academia, as a means of improving Service-based Applications (SBAs). In recent years, a couple of approaches towards monitoring and adaptation of SBAs have been proposed. The aim of this subsection is to analyze these approaches, especially the ones featuring cross-layer and Cloud aspects, and present their main drawbacks. Our focus is on approaches, which deal with both service monitoring and adaptation as these processes are usually coupled in a specific framework or system. Pure service monitoring approaches have been analysed in 4.2.1.2.

The authors in [102] present an approach for self-healing of BPEL processes. This approach is based on the Dynamo [105] monitoring framework along with an AOP extension of ActiveBPEL and a monitoring and recovery subsystem using Drools Event-Condition-Action (ECA) rules. A composition designer provides assertions for invoking, receiving or picking activities in the business process. These assertions can be specified using two domain specific languages (WScOL and WSReL). The problem of selecting alternative services and dealing with possible interface mismatches when forwarding a request to an alternative endpoint recovery is not explicitly addressed. Additionally, the recovery rules cannot be changed dynamically, as they need to be compiled offline.

The VieDAME environment [75] extends the ActiveBPEL engine to enable BPEL process monitoring and partner service substitution based on various strategies. The services are selected according to defined selectors. VieDAME requires service registration to a repository, marking services to be monitored and eventually substituted as replaceable. It uses an engine adapter to extend the engine's functionality, but does not explicitly address fault handling.

The authors in [106] introduce an architecture and a DSL, named MONINA (Monitoring, Integration, Adaptation), that allow to integrate functionality provided by different components and to define monitoring and adaptation functionality. It is similar to FORTH approach, as monitoring is carried out by complex-event processing queries, while adaptation is performed by condition action rules performed. However, it differentiates regarding its scope, which aims at the specification of platforms integrated into a Virtual Service Platform (VSP) that provides a unified view on the functionality of the integrated service platforms that are connected by control interfaces. In addition, it lacks cross-layer and multi-cloud features, as well as experimental analysis of the implemented approach.

4.3.1.1.1 Cross-Layer Approaches

In [107] the authors propose a methodology for the dynamic and flexible adaptation of multi-layer applications using adaptation templates and taxonomies of adaptation mismatches. Templates are exposed as executable BPEL processes that may encapsulate adaptation techniques. The template developers are in charge of associating the templates they develop with adaptation mismatches based on the types of mismatches they can cope with. For each application layer, one or more taxonomies of adaptation mismatches, which may either be generic or contain domain information for particular application domains. The authors use tree-based taxonomies and *is-a* relationship between children and parent mismatches, as well as for the scaled degree of matching between adaptation mismatches. The cross-layer dimension of this approach is achieved by linking adaptation templates, corresponding to layers where adaptation is needed, either directly or indirectly. In the former case, a BPEL adaptation template invokes the WSDL interface of another BPEL adaptation template. In the latter case, a BPEL adaptation template raises an event that will trigger the selection, deployment and execution of another adaptation template. This can be achieved by using standard BPEL activities that are invoked to generate events and receive or pick branches to receive events. Within each layer, the authors assume the availability of several adaptation templates, some of which are linked and which are associated with different taxonomy mismatches.

In [108] the authors present an integrated approach for monitoring and adapting multi-layered SBAs. This approach is based on a variant of MAPE control loops that are typically found in autonomic systems. All the steps in the control loop acknowledge the multi-faceted nature of the system, ensuring that they always reason holistically and adapt the system in a cross-layered and coordinated way. The proposed methodology comprises four main steps: (i) Monitoring and Correlation, where sensors capture run-time data about the software and infrastructural elements, (ii) Analysis of Adaptation Needs, in which the framework identifies anomalous situations and pinpoints where it needs to adapt, (iii) Identification of Multi-layer Adaptation Strategies, in which the framework uses the adaptation capabilities that exist within the system to define a multi-layer adaptation strategy as a set of software and/or infrastructure adaptation actions; and (iv) Adaptation Enactment, where different adaptation engines at the software and infrastructure layer enact their corresponding parts of the multi-layer strategy. This approach comprises a set of mechanisms to provide multi-layer monitoring and adaptation. Its main drawback is that it does not feature proactive adaptation capabilities. In addition, it does not provide in detail how cross-layer monitoring is performed in which the various events are synchronized.

Finally, [109] proposes a holistic SBA management framework, called CLAM, which can deal with cross- and multi-layer adaptation problems. This is achieved in two ways. On the one hand, CLAM identifies the application capabilities affected by the adaptation actions and on the other hand, it identifies an adaptation strategy that solves the adaptation problem by properly coordinating a set of specific adaptation capabilities. This work addresses the cross-layer adaptation problem. The tree-based approach for defining adaptation paths seems very interesting although it can be time-consuming. In addition, during the ranking process of the adaptation branches, cost is not taken into consideration. A drawback of this approach is that it does not elaborate on cross-layer monitoring. Finally, this approach has neither proactive adaptation, nor functional aspects handling capabilities.

4.3.1.2 Languages for Adaptation Plans

As presented in the previous chapter, adaptation efforts mainly on detecting a critical situation and the target state. The description of this state transition is mostly unclear or simplified. Cloudiator targets to be open for various approaches that is why its adaptation component Axe is not tied to a specific language.

In other Cloud orchestration tools such as Apache Brooklyn the rules are simple threshold-based on single metrics. Any more complex rules or event patterns have to be defined and implemented in an external monitoring tool. Axe goes beyond this, as it provides an integrated and easy-to-use solution that even allows changes of the scalability configuration at runtime.

Several projects deal with integrated auto-scaling mechanisms for cloud services. One of them is the EU project CELAR⁶⁶. The language SYBL that specifies elasticity in terms of monitoring, constraints and strategies in multi-level approach describes the adaptation. There are just a few predefined strategy actions, e.g. scale in and scale out, but with the possibility of the specification of user-defined strategies in terms of scripts, which can be called with parameters in a SYBL elasticity description. This is an interesting use-case for Axe, but in respect of CloudSocket, we will aim for an approach that includes (i) a more sophisticated workflow of the actions, and (ii) an awareness of the success of the operation to be able to define fallback strategies, that should lead to a lower violation rate of SLAs.

Bracevac et al. [110] propose the Cloud Platform Language (CPL) that unifies the programming of deployments and applications into a single language as opposed to current provider- and domain-specific languages as e.g. CloudFormation. The adaptation plans that the Cloud Provider Engine will be capable of should be able to cover the main semantics in which the CPL describes such activities: server spawn, snapshot, image replacement, migration and parallelism.

⁶⁶ <http://www.celarcloud.eu/>

In the EPICS EU project, Chen et al. [111] propose to have the auto-scaling system in a self-aware and the decision-making in a distributed way. This means to include the utilization of other services into the own local decision process. Modelling the impact of a scaling action on to the other components, may influence the actual auto-scaling process, e.g. if the replication of a component would heavily increase the communication between another component, the adaptation plan would in this case migrate the latter component to a location near the other component. In addition, the auto-scaling process should be aware of the targeted goal and in case of an opposite results, e.g. undo the scaling action. Another point is the awareness of the interaction, such as that e.g. a scaling action that destroys component x is not executed simultaneously to a scaling action that replicates component x. This awareness should be reflected in the adaptation plans that can be specified in the terms of CloudSocket, in order to assure having less SLO violations.

The OASIS TOSCA standard defines the Cloud applications structure as topology and its management as workflows, so called plans. TOSCA relies on existing languages like BPMN or BPEL to describe those workflows. Kopp et al. [112] propose extensions to BPMN, called BPMN4TOSCA, in order to cater for Cloud-specific tasks and data objects, that eases the use of the language for application modeller. Selecting user-defined actions and allowing specifying branches and gateways for the adaptation plan, is also necessary Adaptation Engine in CloudSocket. Important for CloudSocket is, among the other key benefits of such a workflow approach [113], to be able to cater for fault-handling, and parallelism. Both are crucial for a financially successful and efficient deployment.

The ScalabilityRule Language (SRL) was developed in the course of the PaaSage EU project in order to specify the elasticity behaviour of an application. Concerning the adaptation plan, a scaling rule triggers the execution of an unordered set of scaling actions. This will be improved in the course of the CloudSocket project.

4.3.2 UULM Approach

The adaptation approach of UULM focuses on the auto-scaling of services in a multi-cloud context. As monitoring and adaptation are complementary functionalities, the UULM approach relies on functionalities provided by the Cloud Provider Engine and the UULM monitoring approach. The adaptation framework, namely AXE, is part of the Cloudiator Framework (cf. Figure 19) and processes the aggregated monitoring data (cf. Table 9) of the UULM monitoring approach to enable its auto-scaling capabilities.

4.3.2.1 AXE

UULM's adaptation approach AXE⁶⁷ is the first implementation of the *Scalability Rule Language (SRL)* [4] The concept of SRL was developed in the PaaSage project, amongst others by FORTH and UULM. SRL is a provider-agnostic description language. It provides expressions to define the monitoring raw metric values from VMs and component instances and mechanisms to compose higher-level metrics from raw metrics. Moreover, it comprises mechanisms to express events and event patterns on metrics and metric values. Finally, SRL captures thresholds on the events and actions to be executed when thresholds are violated. A simple SRL rule in prose may be: *add a new instance of this distributed database if (i) all instances have a 5 minute average CPU load > 60%, (ii) at least one instance has a 1 minute average CPU load > 85%, and (iii) the total number of instances is < 6.*

Auto-scaling can be categorised in different classes [114]. SRL, used by AXE, mainly belongs to the threshold-based rules as well as time series analysis class. SRL links a set of threshold-based conditions with each other using binary operators. In addition, any set of thresholds can be linked to the values produced by the metrics. So far, Axe supports the triggering of scale out and scale in actions over application components. Yet, the

⁶⁷ <https://github.com/cloudiator/axe-aggregator>

implementation of further actions, like migration, is an ongoing process. The triggering of rules leads to an invocation of the Cloudiator functionality to bring up a new or shut down an existing VM.

The auto-scaling functionality of AXE builds on top of the monitoring capabilities (cf. section 4.2.3). In particular, any of the conditions connected via Boolean operators is considered a metric on its own taking the values 0 or 1. When the metric value equals to 1, the respective action will be triggered and forwarded as request to the other Cloudiator tools, in particular Colosseum (cf. section 4.1.2). These tasks are executed by the *Scaling Engine* component.

The Scaling Engine (cf. Figure 19) is the central managing environment of AXE that controls the distribution and outsourcing of the computation-heavy work to highly scalable and loosely coupled components, the *Aggregators*. Nevertheless, it is possible to scale the Scaling Engine up to having one instance per scaling rule.

4.3.2.2 Adaptation Plans

As already mentioned, the Cloud Provider Engine employs the AXE tool of Cloudiator, which implemented the SRL and therefore was in the integral state only capable of execution a set of scaling action, defined as scale up and down, but not the execution of more complex, user-defined workflows.

As seen before, adaptation plans are necessary to realize more complex workflows for highly dynamic and distributed applications. This is of great importance for CloudSocket as the target group are the SMEs. They need to benefit of a lightweight IT resource management, as the business process of such an SME lead to very short-term, dynamic workflows, the resources consumption has to be aligned with this business strategy. In order to allow such plans, we extend the current adaptation engine in Cloudiator by the following adaptation items.

The adaptation actions can now be defined as a **sequence**, i.e. each action has a specified order in which the engine will execute it. This caters also for parallelism; since an action can't be executed before, the connected previous actions are finished. An action can be attached to an **alternative plan**, in case it failed. By that, it is possible to change the strategy on run-time. In case no alternative is available, the whole workflow will be rolled back and an error is propagated to the administration. Cool-down interval, migration and user-defined scripts extend the types of actions. The cool-down is the time; the rule engine waits until going over to the next action. Concerning migration, also the life-cycle model has to be extended by *import* and *export* action, which have to be implemented by the user. The return value of *export* is the input parameter of *import*, in terms of a URI. The Colosseum will provide the means of storing data from the entity that exports its data and therefore the URI will link to the home domain of Cloudiator. Still, this is not a fully automatic approach, as the user has to implement the respective actions. The same applies to the user-defined scripts that can be associated to an action of the adaptation plan. By this, it will be possible to have very specific configurations of the Cloud application that are handled throughout all the deployments in an automatic way. The adaptation part of the current interface of the Cloud Provider Engine enables that scaling actions as well as adaptation plans can be **directly executed** and not only by attaching it to a certain condition (threshold to monitoring data). This allows having a convenient management of the scaling also by third-party tools. Adaptation plans, that might **block** each other, are not executed simultaneously. For this, the highly distributed AXE instances will be aware of the on-going activities in other instances. Concerning **service substitution**, it is possible to change the communication of a component towards another component, which results in an extension of the life-cycle actions by a wiring command, which can be called independently in an adaptation plan.

Figure 33 shows an example adaptation plan that vertically scales up a component x, but before that it horizontally scales out the same component in order to have no downtime due to the restarting of the vertically scaled component. This might be necessary, if the user has a very strict SLA concerning downtime. It also shows an alternative plan that, in case the scaling failed, describes the substitution of the service (represented by component x) with another one (here on component z) that is hosted somewhere else and already running. This can be the

case if two services are capable of the same functionality but logically separated for some reasons, e.g. avoid overload or just a component to enable fault-tolerance. In this example component y is for a short time connected to component z, while component x is scaled up.

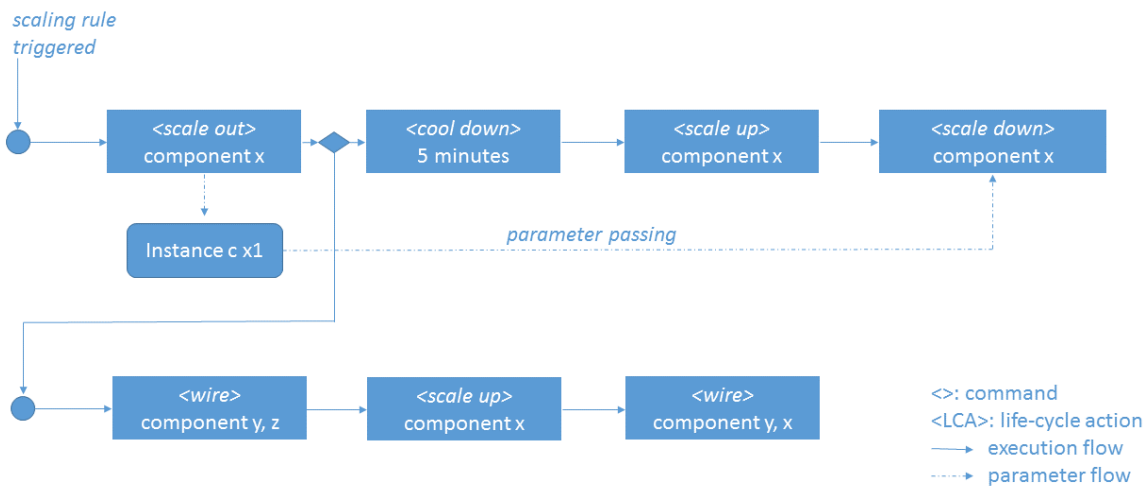


Figure 33 - Example of an adaptation plan with component scaling and short-term service substitution

The concept of adaptation plans is currently work in progress for which the theoretical basis was achieved and worked into the model entities of Cloudiator and later in CAMEL. The implementation and feasibility check is scheduled for the upcoming period of CloudSocket.

4.3.3 FORTH Approach

As already mentioned in section 4.2.4, FORTH has developed a cross-layer monitoring and adaptation framework [99, 114]. By focusing on adaptation, the framework's architecture can be seen in Figure 34. This architecture has been adopted in the context of this project. It comprises mainly seven main components: (a) Rule Engine, (b) Adaptation Engine, (c) Transformer, (d) Rule Derivator and (e-g) layer-specific services (for WFaaS, SaaS, and IaaS layers). The Rule Engine is responsible for detecting which adaptation rules are fired based on the events that have been delivered from the Monitoring Engine. These adaptation rules currently take the form of a mapping between event pattern names to names of adaptation strategies and are specified via the Drools respective language, as Drools is the implementation technology behind this engine. In case two or more rules are fireable, the current practice is to select the one with the highest priority. Such practice could be modified in the near future to more dynamically select the best possible alternative according to the current context. Once the respective adaptation strategy name is identified, then the Adaptation Engine, being a normal Workflow Engine with additional capabilities, is invoked with that name in order to create a specific adaptation workflow instance and execute it. In this sense, there is a fixed mapping between adaptation strategy names and adaptation workflow descriptions. The respective workflow comprises tasks, which map to specific adaptation actions that are layer-specific and map to specific layer-specific services, which deliver the adaptation functionality in each layer. It is the job of the workflow modeller to know which adaptation actions are currently involved in the system/framework in order to specify the respective workflow by mapping the corresponding tasks to those services or pieces of software code that map to these actions.

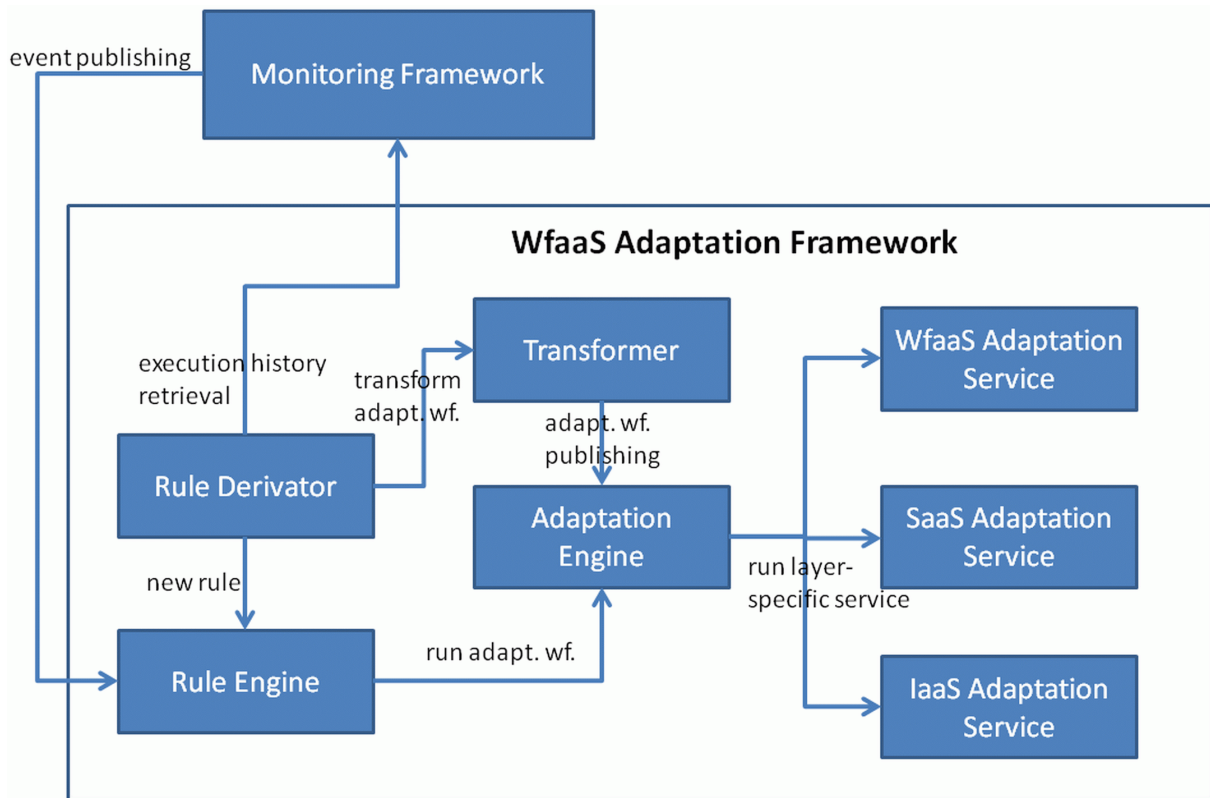


Figure 34 - FORTH's Adaptation Framework

Following the above analysis, there is a need for a set of adaptation services that include the respective adaptation actions needed. In this respect, as the Cloud Provider Engine could be seen as a service, it could be involved in order to perform adaptation actions at the IaaS layer (like scaling or migration ones). Apart from these services, sometimes there will also be a need to include adaptation software code into the Adaptation Engine to be involved in respective automated adaptation workflow tasks. This code is not depicted in Figure 34 as it is considered internal to the Adaptation Engine.

The mapping between event patterns and adaptation strategies is derived in a semi-automatic manner by following a logic-based event pattern discovery approach [118] over the execution history of the BPaaS workflow. This approach is encapsulated in the Rule Derivator. In particular, by considering a set of SLOs that must hold for the BPaaS workflow, the Rule Derivator attempts to find those event patterns that lead to the violation of one or more of these SLOs. As soon as new event patterns are discovered, they are mapped to specific adaptation strategies that need to be performed to alleviate the respective SLO violations. Such strategies are derived in a semi-automatic manner via simple adaptation rules that are manually provided by the expert. These simple rules take the form of a mapping between single events to one adaptation action to be performed to address it. Then, by considering the set of events included in the respective event pattern discovered, the corresponding adaptation actions mapping to these events are combined in order to produce the relevant adaptation strategy. The space of possible action combinations can be filtered by knowing which actions have similar effects or contradictory ones and which actions can be parallelised or executed only in sequence. In the end, actually a set of adaptation strategies are derived, as workflows of adaptation actions, that need to be selected in order to specify the more complex adaptation rule. The selection currently mainly relies on considering the priorities put on the simple adaptation rules from which the more complex candidate ones have been derived. The candidate adaptation rule with the highest multiplication of involved priorities is selected. Please note that apart from the final generation and modelling of the complex adaptation rule, the respective workflow has to be constructed mapping to its name. In this respect, the internal structure of the adaptation strategy, as produced from the Rule Derivator, has to be automatically transformed into a workflow. This is actually done by the Transformer.

4.3.4 Integration / Synergy of Approaches

By considering the analysis of the two adaptation approaches in the previous two sections, we regard that these approaches are more or less complementary. In fact, one of the approach could be considered as part of the other. In particular, as the Cloud Provider Engine currently covers the IaaS layer and the performance of scaling plus migration actions on this layer, then it can be considered as a composite service that offers the respective adaptation functionality in this layer. By following then the approach of FORTH, we could exploit this service along with others provided in the rest of the layers in order to completely cover complex adaptation scenarios across all the possible layers.

To enable this composition of approaches, the Cloud Provider Engine should be offered as a service that needs to expose the respective adaptation functionality. The current prototype of FORTH does not need to be heavily modified apart from the fact that it also needs to be offered as a service that could be exploited by a BPaaS Execution Environment. However, also other details need to be fixed which can have an effect on the respective prototype implementation code. Section 2.4.1 highlights the need for introducing a specific adaptation rule DSL as a sub-DSL of CAMEL which will enable the specification of the cross-layer adaptation behaviour to handle advanced adaptation scenarios. As such, once this extension is in place, it will have to be adapted by the adaptation approach of FORTH. One thought of how this could be performed would be to modify the Rule Engine in order to be able to process CAMEL adaptation rules. Here we have mainly two possibilities: (a) the Rule Engine implementation is modified - for instance, it could be argued that there is no need for an actual rule engine but for a system which can identify which rules are triggered based on specific events. Different techniques could then be exploited like the ones employed by the UULM approach in the case of scaling rule triggering; (b) the Rule Engine implementation is not modified but an additional component is added in the overall architecture responsible for translating CAMEL adaptation rules to Drools rules. Apart from possibly adapting the Rule Engine, the Transformer might also need to be adapted such that it is able to transform the action parts in the CAMEL adaptation rules to respective adaptation workflows.

4.3.5 Future research

Apart from the pending combination of FORTH and UULM adaptation approaches to cover a complete cross-layer BPaaS adaptation, there are also certain research directions that are worth investigating, followed and implemented in the respective research prototypes. These directions are analysed in the following sub-sections.

4.3.5.1 *Dynamic Adaptation Workflow Concretisation*

A system may not have stable adaptation capabilities. Systems evolve and respective capabilities have to be updated and expanded. As such, by considering the case of adaptation workflows, it is better not to be fixed but dynamically derived on demand, when the need to perform the respective adaptation is raised, based on the current adaptation capabilities of the system. In this sense, the same adaptation rule can be realised in different ways at different time points and different realisations might lead to better adaptation performance. Performance does matter as the longer an adaptation workflow takes to finish, the higher is the risk that the respective adaptation actions are not performed in time such that the corresponding SLO violation is avoided.

Concerning the way the abstract to concrete adaptation workflow concretisation can be achieved, the same logic can be followed as in the case for abstract BPaaS workflow allocation via exploiting semantics to accurately discover those actions that map to a certain adaptation workflow task. Moreover, the same techniques as in (composite) service concretisation can be exploited to select the best alternatives for each workflow task in order to satisfy global quality constraints overall adaptation workflow.

4.3.5.2 Optimised Derivation of Adaptation Strategies

While the priority of an adaptation action over the handling of a specific event can enable the respective composition of actions to formulate an adaptation strategy, it still maps to a subjective approach as such priorities are given by the rule expert and can reflect only common situations. In this sense, a more dynamic approach must be followed which handles the cold start problem via the original approach but then dynamically modifies the adaptation strategy selection decisions according to actual runtime/execution knowledge. For example, if one adaptation strategy consistently fails to remedy the problem (e.g., a SLO violation) to be solved, then an alternative strategy needs to be employed. Moreover, by considering individual actions in each strategy, similar derivations can be reached about which individual actions should be preferred over the others. As such, by following the execution knowledge, the system reaches more optimal points, which enable it to better address the various problematic situations that occur or are about to occur. Relevant work on this subject includes [88], which could be adopted in the context of the project and the respective frameworks that are exploited.

4.3.5.3 Layer-Specific Adaptation Action Realisation

To address complex adaptation situations in a cross-layer manner, respective adaptation functionality in each layer must be in place. Thus, there is a need to equip the adaptation system with a collection of such functionalities and advance research over particular types of functionalities. For example, while horizontal scaling is well supported by both existing research and commercial prototypes, this is not the case for stateful component migration. As such, the research work currently performed in UULM attempts to address this kind of migration for specific types of stateful components, i.e., databases, is a very nice starting point.

Concerning the SaaS level, existing functionality delivered by FORTH already exists including components supporting dynamic service discovery and functional as well as non-functional service composition. Such components could then be easily offered as a service and included in the respective combined adaptation system. Some of these components might need to be slightly updated, e.g., the service composition to adapt just a part of a currently running workflow and not the whole workflow.

Concerning the WaaS level, research work, to be adopted, has focused on addressing mainly the instance migration problem. However, the workflow re-composition problem should be also addressed as is relevant in different cases: (a) critical service functionality mapping to one or more tasks ceases to exist; (b) non-functional requirements are changed and this leads to no possibility for concretising the current structure of the workflow. Similar principles as in service re-composition could apply but the issue here is that we are dealing with a different level with each own peculiarities. Fortunately, the same or similar problems as in the determination of an abstract workflow in the context of a BPaaS also apply here.

5 INTERACTION WITH OTHER ENVIRONMENTS

5.1 Required Input

The previous sections have shed light on particular research items or approaches that have been mainly pursued by the research partners of this project. However, these approaches or items require particular forms of input in order to function as expected. In the sequel, we focus on each of the items and indicate the kind of input that is required from those environments that map either to a phase before the allocation one (i.e., the Design Environment) or to cross-environment functionality (like the registries offered by the Marketplace).

Smart service discovery and functional composition. As indicated in section 3.2, the respective algorithms proposed work over OWL-S and OWL-Q semantic functional and non-functional specifications, respectively. However, they could be modified to obtain respective input from a semantic or semantically annotated service repository. In this case, the major issue here is to have some sorts of semantics that accompany the description of services whichever is the place on which these descriptions are stored. In this sense, what is actually expected from these algorithms is a semantically-enhanced service repository which can then exploit in order to produce their own structures that assist in the speed-up of the service discovery and composition processes.

Non-Functional Service Composition / Concretisation. The respective algorithm needs to have a semantically annotated workflow structure covering all possible requirements posed over this structure. These semantic annotations are needed for service discovery purposes, i.e., to discover those services that functionally and non-functionally match a particular workflow task. They are also needed in order to properly format the optimisation problem on which the service concretisation algorithm relies. As already indicated in the previous paragraph, the respective service discovery and composition algorithms could function even in the non-presence of specific semantic service description formalisms. However, there is a need to deliver a formalism which enables the semantic annotation of BPaaS workflows in a global and local (task) level. Such an annotation can rely on the different annotation mechanisms as indicated in D3.1. The respective tools support to produce these annotations is already in place as indicated in D3.2 (with the sole exception of OWL-Q annotations that are still not possible). Functional annotations should rely on the use of domain concepts while non-functional annotations should rely on the use of semantic quality models which need to be specified via OWL-Q. Please note that these annotations should concern the technical and not the business level. This means that, for instance, non-functional annotations can refer to more technical quality terms rather than business ones (e.g., workflow processing time instead of business process duration - while it is apparent in this case that the latter is equal to the former). This is suitable in order to be able not only to perform BPaaS workflow concretisation but also assist in the specification of SLAs as well as for the subsequent BPaaS workflow monitoring. Concerning functional annotations, different ontologies might be involved with respect to the business level. In particular, a business process model can be annotated via concepts representing business objects while a workflow model should be annotated via concepts representing more technical concepts.

Monitoring Approaches. Semantic annotation on technical non-functional requirements should be in place. This is exactly what is also demanded by the previous algorithm. The main issue, however, here is the compositionality of the quality terms as this enhances the monitorability of the requirements. In particular, while technical requirements at the workflow level can include high-level quality terms like metrics, there should be a way to decompose these terms into terms that can be computed in case that a sensor is not available for the measurement of the high-level quality term. To assist in this matter, it is suitable to have a semantic metric repository/registry, relying on OWL-Q, via which high-level quality terms are completely defined, including the ways that they can be computed from lower-level metrics. Apart from this, in terms of technical requirements that are specified via CAMEL, we need to stress here the need to enable semantic annotations on the CAMEL's metric meta-model. This can be of course

considered as an internal (based on the context of this deliverable) input requirement that is demanded by the Execution Environment and has to be provided by the Allocation Environment.

Deployment Plus Adaptation Approaches. Both deployment and adaptation need to benefit from existing deployment/selection and adaptation rules that are provided by experts. Such rules could be based on a high-level specification language like DMN and be provided by the Design to the Allocation Environment. Then, by following the approach sketched in section 3.4, DMN specifications can be mapped to adaptation rules specified in CAMEL as well as to deployment knowledge (rules) that can assist in the production of the most suitable deployment plan. While not yet explicated, deployment rules could be specified in CAMEL or via any other formalism. They could also lead to a small extension of the BPaaS workflow concretisation algorithm in order to transform them into respective constraints of the optimisation problem to be solved. However, another use for them would be to enable more dynamic deployment scenarios where some deployment logic is not concretely specified in the deployment plan but has to be concretised during deployment by the Cloud Provider Engine.

Deployment Approach. The Cloud Provider Engine requires the existence of a service registry which can indicate the services available at the IaaS and PaaS layers. Such a registry could be exploited to drive the dynamic deployment behaviour based on the aforementioned scenarios in the previous paragraph. Moreover, it could also be the case that the deployment plan refers to respective entries of this registry from which the Cloud Provider Engine will obtain information that can support the instantiation of the respective components (software or VM). This would lead to a more lightweight approach in deployment plan specification.

5.2 Exploitable Output

Based on the current logical interaction order between the different environments, it is apparent that the Allocation Environment produces output that is mainly exploited by the Execution Environment in an indirect manner (i.e., after the purchase of a specific bundle). However, there is an additional case which needs to be accounted. This concerns the fact that the Allocation Environment would not be able to define a BPaaS bundle out of a specific corresponding design package due to various technical reasons. These can include: (a) over-constrained requirements at the business level that cannot be satisfied at the technical level (e.g., performance requirements not met by any service composition); (b) missing technical requirements or technical observations/facts that could also influence business decisions (e.g., high cost of the bundle). In such a case, an interaction with the Design Environment should take place in order to either change the high-level requirements or to provide new ones to cover the missing technical ones. We can possibly consider either a kind of notification mechanism employed in order to inform the Design Environment about this problematic case and enable the adjustment of the requirements or the interchangeable usage of the environments by the same individual (e.g., a technical expert, hired by the BPaaS broker, involved in the design of the workflow and its allocation).

The main exploiter of the Execution Environment is the Evaluation one which needs to retrieve the most suitable knowledge from the execution of one or more BPaaS in order to appropriately and properly perform the respective analysis tasks. To assist in the different types of analysis, we foresee the following implications:

- *direct or indirect semantic annotation of measurements:* either measurements have to be directly annotated with the respective metric that they refer to or they need to point to a metric specification in which semantic annotations have been included. Through this semantic annotation, the semantic lifting of measurements can be achieved and their exploitation by the semantic KPI analysis approach envisioned for the Evaluation Environment.
- *other BPaaS execution information:* some of the remaining analysis mechanisms of the Evaluation Environment need to derive best deployment plans for BPaaS and discrepancies in BPaaS workflows. To enable the proper functioning of such mechanisms, respective information should be made available by the Execution Environment which ideally would be nice if it is already semantically annotated. Such

information spans: (a) workflow log files from which we can inspect all workflow tasks that have been performed, the timing of their execution and the respective order; (b) SaaS, IaaS and PaaS logs from which we can derive information about when services executed and how long did their execution last; (c) live allocation information mapping component instances to each other all the way up until the workflow level (such that we know which IaaS instances were used to host which internal software component instances which realise the functionality of which task instances in the BPaaS workflow). While (a) can be easy to gather as most workflow execution engines do provide or can be configured to provide such logs, it is not the case for the rest of the information. Thus, the respective components in the Execution Environment should either already offer or be enhanced to offer such information.

6 SUMMARY: RESEARCH SHOWROOM

This chapter provides a brief summary of all research blueprints presented in this deliverable. Each research blueprint is categorized in respect to its current state, while in the end the handover process for the respective research assets of the blueprints to WP4 is outlined. This chapter concludes with an overall summary and outlook to the forthcoming research.

6.1 Research assets

In the following, the research assets of the respective Blueprint categories (BPaaS Modelling Blueprint, Allocation Environment Blueprint, & Execution Environment Blueprint) are briefly analysed. This analysis will ease the handover process to WP4 as it provides a solid overview and maturity level for each asset to the consortium. Each asset analysis includes a short *summary* with the focus on the added value, the *asset type* and the *research state*. The asset type defines the relation to existing components of the CloudSocket. Possible types are: *new asset*, *enhancement of component X* or *replacement of component X*. The research state indicates the actual state of the asset and the estimated time in months to provide a first prototype, *estimated time to prototype (ETTP* in months). Possible research states are idea (ETTP ~ 12), concept (ETTP ~ 9), in process (ETTP ~ 6) and alpha version (~3). In order to ease the evaluation and also have an indication of the integration effort required for WP4, attributes like the existing/targeted license and the dependencies to existing components are also included (along with an explanation of why these dependencies hold). The analysis over all research assets for each Blueprint category is incarnated in the following tables.

	BPaaS Modelling Blueprint
Name	1. PaaS/SaaS support of CAMEL (cf. chapter 2.2.2)
Summary	Provide a cloud service level agnostic modelling approach for services
Dependencies	Allocation Environment, Cloud Provider Engine
Asset Type	Extension of CAMEL
Research State	In process
License	Open-Source
Name	2. SLA support in OWL-Q (cf. chapter 2.3.2)
Summary	Extend OWL-Q to support the semantic specification of SLA (templates)
Asset Type	Extension of OWL-Q
Research State	Alpha version
License	Open-Source

Table 11 - BPaaS Modelling Blueprint assets

	Allocation Environment Blueprint
Name	3. Smart Service Discovery and Composition Tools (cf. chapter 3.2 and 3.3)
Summary	Semantic functional and non-functional service discovery and composition tools enabling automatic mapping of abstract to concrete BPaaS workflows
Dependencies	Registries (service and provider/laaS/PaaS)
Dependency Explanation	Cloud Service Offerings must be specified in the registries. Moreover, semantic annotations should be in place for specific types of cloud services. The tools need to be extended in order to be able to operate over these annotations and not just service specifications conforming to a specific semantic description language
Asset Type	New asset
Research State	Alpha version
License	Open-Source
Name	4. DMN to CAMEL Mapping (cf. chapter 3.4)
Summary	Semi-automatic generation of CAMEL based on business values
Dependencies	CAMEL, Registries
Dependency Explanation	The dependency to registries is required in order to be able to map high-level decisions to low-level ones which map to the selection of particular services that are fully described in the registries. The conditions over service selection will also rely on metrics that are defined in the metric registry.
Asset Type	New asset
Research State	Idea
License	Open Source

Table 12 - Allocation Environment Blueprint assets

	Execution Environment Blueprint
Name	5. PaaS orchestration and abstraction layer (cf. chapter 4.1.2.3)
Summary	Enabling Multi-PaaS orchestration by abstracting PaaS provider specific characteristics
Dependencies	Cloud Provider Engine
Asset Type	Extension to Cloud Provider Engine
Research State	In process
License	Open-Source

Name	6. Dynamic IaaS Selection at Runtime (cf. chapter 4.1.3.1)
Summary	Dynamic selection of IaaS to host internal BPaaS components based on different criteria like tenant location
Dependencies	Registries
Dependency Explanation	IaaS registry needs to be populated accordingly such that this algorithm can really function as expected and provide respective results
Asset Type	New asset
Research State	Idea
Name	7. Distributed and self-scalable Monitoring Architecture (cf. chapter 4.2.3)
Summary	Provide self-scaling Monitoring Architecture with a flexible TSDB storage engine and customisable sensors
Dependencies	Metric Registry, CAMEL/OWL-Q (BPaaS bundle)
Asset Type	Enhancement of Monitoring Engine
Research State	In process
License	Open-Source
Name	8. Cross-Layer Monitoring Framework (cf. chapter 4.2.4)
Summary	Cross-layer monitoring framework for BPaaS which provides measurements on metrics at different layers of abstraction
Dependencies	Metric Registry, CAMEL/OWL-Q (BPaaS bundle)
Dependency Explanation	Need to know the metrics that need to be sensed or aggregated as well as the components whose properties are measured by these metrics. The Metric Registry provides the specification of the metrics but there is also a need for having access to the BPaaS bundle description and especially the monitoring part in CAMEL such that we have the knowledge about the respective conditions and metric contexts that have to be accommodated.
Asset Type	Replacement of Monitoring Engine
Research State	In Process
Licence	Open-Source
Name	9. Synergic Cross-Layer Monitoring Framework (cf. chapter 4.2.6.3)
Summary	Cross-layer monitoring framework produced by combining the monitoring frameworks from FORTH and UULM
Dependencies	Metric Registry, CAMEL/OWL-Q (BPaaS bundle)

Dependency Explanation	Need to know the metrics that need to be sensed or aggregated as well as the components whose properties are measured by these metrics. The Metric Registry provides the specification of the metrics but there is also a need for having access to the BPaaS bundle description and especially the monitoring part in CAMEL such that we have the knowledge about the respective conditions and metric contexts that have to be accommodated.
Asset Type	Enhancement of Monitoring Engine
Research State	Concept
Licence	Open-Source
Name	10. AXE Adaptation Framework
Summary	An adaptation framework supporting the Scalability Rule Language for enabling complex BPaaS adaptations on the IaaS level
Dependencies	CAMEL (BPaaS bundle), Cloud Provider Engine, Metric Registry, CAMEL/OWL-Q (BPaaS bundle)
Asset Type	Enhancement of Adaptation Engine
Research State	Alpha version
Licence	Open-Source
Name	11. Cross-Layer Adaptation Framework (cf. Chapter 4.3.3)
Summary	An adaptation framework for BPaaS enabling to perform adaptation strategies in a cross-layer manner to resolve respective problematic situations
Dependencies	CAMEL (BPaaS bundle), Component Registries
Dependency Explanation	Need to know what are the adaptation strategies that have to be triggered in terms of adaptation rules and this information will be available in the forthcoming CAMEL extension (see section 2.4.1). In addition, if adaptation actions are considered as software components, then we need their description in the software component registry.
Asset Type	Replacement of Adaptation Engine
Research State	Concept
Licence	Open-Source
Name	12. Synergic Cross-Layer Adaptation Framework (cf. chapter 4.3.4)
Summary	Adaptation framework produced from the combination of the adaptation frameworks of FORTH and UULM.
Dependencies	CAMEL (BPaaS bundle), Component Registries

Dependency Explanation	Need to know what are the adaptation strategies that have to be triggered in terms of adaptation rules and this information will be available in the forthcoming CAMEL extension (see section 2.4.1). In addition, if adaptation actions are considered as software components, then we need their description in the software component registry.
Asset Type	Enhancement of Adaptation Engine
Research State	Concept
Licence	Open-Source

Table 13 - Execution Environment Blueprint assets

6.2 Blueprint handover process

Some of the presented research blueprints and assets may be selected to be integrated into the stable CloudSocket architecture through WP4. The blueprint/asset analysis in the previous section facilitates the hand over process of the respective blueprints. The handover process covering both D3.3 and D3.4 is depicted in Figure 35. All described assets are presented to the whole consortium, especially the WP4 stakeholders. With continuous demonstrations along general assemblies and remote session an agile interaction with the end users is achieved. This interaction passes the initial presentation of the research ideas and WP4 provided an initial feedback. Further, the results of D3.3 are presented in order to derive a first prioritization of the blueprints from a WP4 perspective. This provided overview of the ongoing research blueprints can then be already considered for the upcoming Deliverable D4.5 "Final CloudSocket Architecture" which is due in M21. As not all research blueprints might be considered with a high priority, the involved WP3 partners have to take the decision, which blueprints they will follow in order to provide prototypes.

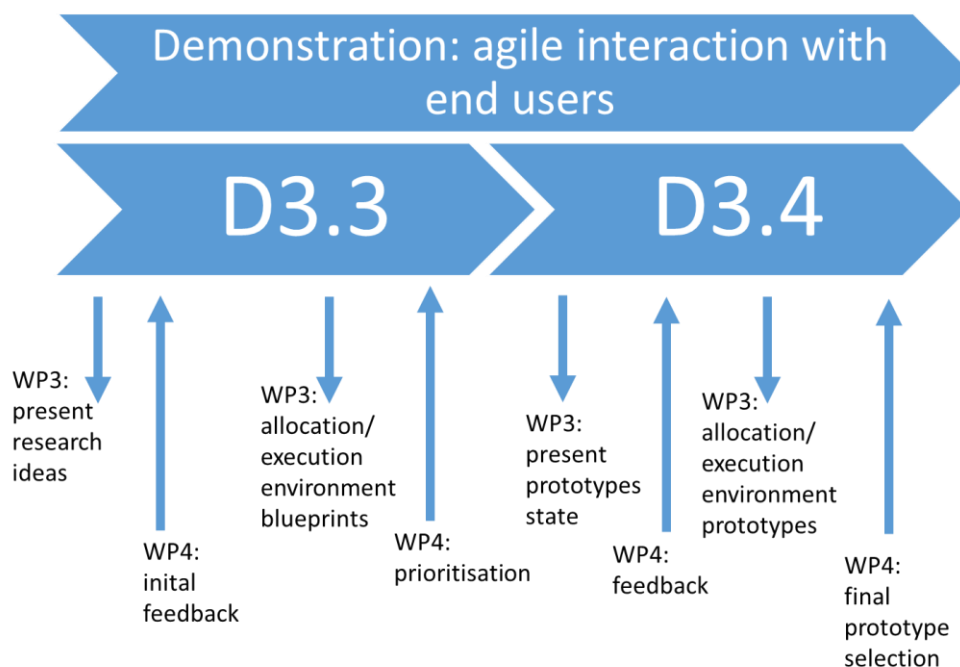


Figure 35 - Research blueprint handover process

Based on the feedback from WP4, the focus will be placed on the higher prioritized research assets in order to present the progress of respective blueprint prototypes during upcoming demonstrations. During the prototype development process, WP4 is able to monitor it via periodic conference calls and technical workshops.

6.3 Summaray and Future Work

This document comprises the mapping and execution from higher level business processes and workflows to deployable BPaaS Bundles. Therefore, the three Blueprint categories, BPaaS Modelling, BPaaS Allocation Environment Blueprints and BPaaS Execution Environment Blueprints are presented. The identified research challenges (cf. section 1.2) are addressed for each Blueprint category by the research assets that have been presented in the previous section 6.1.

The developed research assets of each Blueprint category are evaluated and prioritised by WP4. This allows the WP3 to focus and push the most beneficial assets in each Blueprint for CloudSocket in order to provide deployable prototypes in the context of D3.4 “BPaaS Allocation and Execution Environment Prototypes”. Further, the presented Blueprints allow a smooth transition into the D3.5 “BPaaS Monitoring and Evaluation Blueprints” by providing respective input data and interfaces facilitating the harvesting of such data.

7 REFERENCES

- [1] T. C. consortium, 'D4.1 – First CloudSocketArchitecture', CloudSocket project deliverable, 2015.
- [2] D. Palma, M. Rutkowski, and T. Spatzier, *TOSCA Simple Profile in YAML Version 1.0*. 2015.
- [3] N. Ferry, F. Chauvel, A. Rossini, B. Morin, and A. Solberg, 'Managing multi-cloud systems with CloudMF', in *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*, 2013, pp. 38–45.
- [4] K. Kritikos, J. Domaschka, and A. Rossini, 'SRL: A ScalabilityRule Language for Multi-Cloud Environments', in *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, 2014, pp. 1–9.
- [5] T. P. Consortium, 'D2.1.2 – CloudML Implementation Documentation (First version)', PaaSage project deliverable, 2014.
- [6] K. Kritikos, P. Plebani, and D. Plexousakis, 'Semantic qos metric matching', in *Cloud Forward Conference*, 2015.
- [7] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, and others, *Web services description language (WSDL) 1.1*. 2001.
- [8] M. J. Hadley, 'Web application description language (WADL)', 2006.
- [9] O. Coalition, *OWL-S 1.0 Release*. DAML, At <http://www.daml.org/services/owl-s/1.1>, 2005.
- [10] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel, 'Web service modeling ontology', *Appl. Ontol.*, vol. 1, no. 1, pp. 77–106, 2005.
- [11] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut, 'Quality of service for workflows and web service processes', *Web Semant. Sci. Serv. Agents World Wide Web*, vol. 1, no. 3, pp. 281–308, 2004.
- [12] J. Amsden, 'Modeling with soaml, the service-oriented architecture modeling language—part 1—service identification', *IBM Dev. Works Httpwww Ibm Comdeveloperworksrationallibrary09modelingwithsoaml-1index Html*, 2010.
- [13] D. Jordan, J. Evdemon, A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, and others, *Web services business process execution language version 2.0*, vol. 1. 2003.
- [14] B. P. Model, 'Notation (BPMN) version 2.0', *OMG Specif. Object Manag. Group*, 2011.
- [15] K. Hinkelmann, K. Kritikos, S. Kurjakovic, B. Lammel, R. Woitsch, and M. Albbayrak, 'D3.1 – Modelling Framework for BPaaS', CloudSocket project deliverable, 2016.
- [16] K. Kritikos, B. Pernici, P. Plebani, C. Cappiello, M. Comuzzi, S. Benrenou, I. Brandic, A. Kertész, M. Parkin, and M. Carro, 'A survey on service quality description', *ACM Comput. Surv. CSUR*, vol. 46, no. 1, p. 1, 2013.
- [17] D. K. Nguyen, F. Lelli, Y. Taher, M. Parkin, M. P. Papazoglou, and W.-J. van den Heuvel, 'Blueprint template support for engineering cloud-based services', in *European Conference on a Service-Based Internet*, 2011, pp. 26–37.
- [18] DMTF, *Cloud Infrastructure Management Interface (CIMI) Model and RESTful HTTP-based Protocol*. 2013.
- [19] Open Grid Forum, *Open Cloud Computing Interface - Core*. 2011.
- [20] F. Moscato, R. Aversa, B. D. Martino, T.-F. Fortis, and V. Munteanu, 'An analysis of mosaic ontology for cloud resources annotation', in *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on*, 2011, pp. 973–980.
- [21] G. S. Blair, N. Bencomo, and R. B. France, 'Models@run.time', *IEEE Comput.*, vol. 42, no. 10, pp. 22–27, 2009.
- [22] C. Quinton, D. Romero, and L. Duchien, 'Cardinality-based feature models with constraints: a pragmatic approach', in *Proceedings of the 17th International Software Product Line Conference*, 2013, pp. 162–166.
- [23] K. Kritikos and D. Plexousakis, 'Semantic qos metric matching', in *Web Services, 2006. ECOWS'06. 4th European Conference on*, 2006, pp. 265–274.
- [24] K. Kritikos, P. Plebani, and D. Plexousakis, 'Semantic qos metric matching', in *Cloud Forward Conference*, 2015.
- [25] M. Klusch, B. Fries, and K. Sycara, 'Automated semantic web service discovery with OWLS-MX', in *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, 2006, pp. 915–922.
- [26] K. Kritikos and D. Plexousakis, 'Novel Optimal and Scalable Nonfunctional Service Matchmaking Techniques', *Serv. Comput. IEEE Trans. On*, vol. 7, no. 4, pp. 614–627, 2014.

- [27] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M.-C. Shan, 'Adaptive and dynamic service composition in eFlow', in *Advanced Information Systems Engineering*, 2000, pp. 13–31.
- [28] A. Brogi and S. Corfini, 'Ontology-and behavior-aware discovery of Web service compositions', *Int. J. Coop. Inf. Syst.*, vol. 17, no. 3, pp. 319–347, 2008.
- [29] P. Bertoli, M. Pistore, and P. Traverso, 'Automated composition of web services via planning in asynchronous domains', *Artif. Intell.*, vol. 174, no. 3, pp. 316–361, 2010.
- [30] C. Zeng, X. Guo, W. Ou, and D. Han, 'Cloud computing service composition and search based on semantic', in *Cloud Computing*, Springer, 2009, pp. 290–300.
- [31] K. Kofler, I. ul Haq, and E. Schikuta, 'User-centric, heuristic optimization of service composition in clouds', in *Euro-Par 2010-Parallel Processing*, Springer, 2010, pp. 405–417.
- [32] Q. He, J. Han, Y. Yang, J. Grundy, and H. Jin, 'QoS-driven service selection for multi-tenant SaaS', in *Cloud computing (cloud), 2012 IEEE 5th international conference on*, 2012, pp. 566–573.
- [33] E. Wittern, J. Kuhlenskamp, and M. Menzel, 'Cloud service selection based on variability modeling', in *Service-Oriented Computing*, Springer, 2012, pp. 127–141.
- [34] A. Klein, F. Ishikawa, and S. Honiden, 'Towards network-aware service composition in the cloud', in *Proceedings of the 21st international conference on World Wide Web*, 2012, pp. 959–968.
- [35] A. M. Ferreira, K. Kritikos, and B. Pernici, 'Energy-aware design of service-based applications', in *Service-Oriented Computing*, Springer, 2009, pp. 99–114.
- [36] X. Zhao, L. Shen, X. Peng, and W. Zhao, 'Toward SLA-constrained service composition: An approach based on a fuzzy linguistic preference model and an evolutionary algorithm', *Inf. Sci.*, vol. 316, pp. 370–396, 2015.
- [37] G. Horn, 'A vision for a stochastic reasoner for autonomic cloud deployment', in *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*, 2013, pp. 46–53.
- [38] K. E. Kritikos, 'Qos-based web service description and discovery', University of Crete, 2008.
- [39] K. Kritikos and D. Plexousakis, 'Requirements for QoS-based web service description and discovery', *Serv. Comput. IEEE Trans. On*, vol. 2, no. 4, pp. 320–337, 2009.
- [40] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, 'Pellet: A practical owl-dl reasoner', *Web Semant. Sci. Serv. Agents World Wide Web*, vol. 5, no. 2, pp. 51–53, 2007.
- [41] B. Blake, L. Cabral, B. König-Ries, U. Küster, and D. Martin, *Semantic web services: Advancement through evaluation*. Springer Science & Business Media, 2012.
- [42] 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.
- [43] G. Baryannis and D. Plexousakis, 'WSSL: a fluent calculus-based language for web service specifications', in *Advanced Information Systems Engineering*, 2013, pp. 256–271.
- [44] K. Kritikos, P. Plebani, and D. Plexousakis, 'Semantic qos metric matching', in *Cloud Forward Conference*, 2015.
- [45] T. L. Saaty, *The Analytic Hierarchy Process, Planning, Priority Setting, Resource Allocation*. New York: McGraw-Hill, 1980.
- [46] O. M. Group, 'Decision Model and Notation', OMG, <http://www.omg.org/spec/DMN/1.1/>, 2015.
- [47] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, 'Above the clouds: A Berkeley view of cloud computing', *Dept Electr. Eng Comput Sci. Univ. Calif. Berkeley Rep UCBECS*, vol. 28, no. 13, p. 2009, 2009.
- [48] D. Baur, D. Seybold, F. Griesinger, A. Tsitsipas, C. B. Hauser, and others, 'Cloud Orchestration Features: Are Tools Fit for Purpose?', in *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, 2015, pp. 95–101.
- [49] P. Mell and T. Grance, 'The NIST definition of cloud computing', 2011.
- [50] J. Domaschka, D. Baur, D. Seybold, and F. Griesinger, 'Cloudiator: a cross-cloud, multi-tenant deployment and runtime engine', in *9th Symposium and Summer School on Service-Oriented Computing*, 2015.
- [51] E. Hossny, S. Khattab, F. A. Omara, and H. Hassan, 'Semantic-based generation of generic-API adapters for portable cloud applications', in *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, 2016, p. 1.
- [52] E. Alomari, A. Barnawi, and S. Sakr, 'Cdport: A portability framework for nosql datastores', *Arab. J. Sci. Eng.*, vol. 40, no. 9, pp. 2531–2553, 2015.
- [53] E. Hossny, S. Khattab, F. Omara, and H. Hassan, 'Towards a standard PaaS implementation API: A standard cloud persistent-storage API', in *3rd International IBM Cloud Academy Conference ICACON 2015*, Budapest, Hungary, 2015.

- [54] M. Sellami, S. Yangui, M. Mohamed, and S. Tata, 'PaaS-independent Provisioning and Management of Applications in the Cloud', in *2013 IEEE Sixth International Conference on Cloud Computing*, 2013, pp. 693–700.
- [55] S. Walraven, D. Van Landuyt, A. Rafique, B. Lagaisse, and W. Joosen, 'PaaS Hopper: Policy-driven middleware for multi-PaaS environments', *J. Internet Serv. Appl.*, vol. 6, no. 1, pp. 1–14, 2015.
- [56] 'OCCI Working Group'. [Online]. Available: <http://occi-wg.org/>. [Accessed: 08-Apr-2016].
- [57] D. Palma, M. Rutkowski, and T. Spatzier, *TOSCA Simple Profile in YAML Version 1.0*. 2015.
- [58] A. Karmarkar, 'CAMP: a standard for managing applications on a PaaS cloud', 2014, pp. 1–2.
- [59] J. Domaschka, F. Griesinger, D. Baur, and A. Rossini, 'Beyond Mere Application Structure Thoughts on the Future of Cloud Orchestration Tools', *Procedia Comput. Sci.*, vol. 68, pp. 151–162, 2015.
- [60] H.-L. Truong, R. Samborski, and T. Fahringer, 'Towards a framework for monitoring and analyzing qos metrics of grid services', in *e-Science and Grid Computing, 2006. e-Science'06. Second IEEE International Conference on*, 2006, pp. 65–65.
- [61] R. Sakellariou and V. Yarmolenko, 'Job Scheduling on the Grid: Towards SLA-Based Scheduling.', in *High Performance Computing Workshop*, 2006, pp. 207–222.
- [62] C. Cappiello, K. Kritikos, A. Metzger, M. Parkin, B. Pernici, P. Plebani, and M. Treiber, 'A quality model for service monitoring and adaptation', in *Workshop on Service Monitoring, Adaptation and Beyond*, 2009, p. 29.
- [63] K. Kritikos and D. Plexousakis, 'Mixed-integer programming for QoS-based web service matchmaking', *Serv. Comput. IEEE Trans. On*, vol. 2, no. 2, pp. 122–139, 2009.
- [64] N. B. Mabrouk, N. Georgantas, and V. Issarny, 'A semantic end-to-end QoS model for dynamic service oriented environments', in *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, 2009, pp. 34–41.
- [65] X. Zheng, P. Martin, K. Brohman, and L. Da Xu, 'CLOUDQUAL: a quality model for cloud services', *Ind. Inform. IEEE Trans. On*, vol. 10, no. 2, pp. 1527–1536, 2014.
- [66] A. K. Bardsiri and S. M. Hashemi, 'Qos metrics for cloud computing services evaluation', *Int. J. Intell. Syst. Appl. IJISA*, vol. 6, no. 12, p. 27, 2014.
- [67] Y. Li and S. Manoharan, 'A performance comparison of SQL and NoSQL databases', in *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*, 2013, pp. 15–19.
- [68] S. K. Garg, S. Versteeg, and R. Buyya, 'SMICloud: a framework for comparing and ranking cloud services', in *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, 2011, pp. 210–218.
- [69] N. R. Herbst, S. Kounev, and R. H. Reussner, 'Elasticity in Cloud Computing: What It Is, and What It Is Not.', in *ICAC*, 2013, pp. 23–27.
- [70] M. Beltran, 'Defining an Elasticity Metric for Cloud Computing Environments', in *Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools*, 2016, pp. 172–179.
- [71] S. Islam, K. Lee, A. Fekete, and A. Liu, 'How a consumer can measure elasticity for cloud platforms', in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, 2012, pp. 85–96.
- [72] S. Lehrig, H. Eikerling, and S. Becker, 'Scalability, Elasticity, and Efficiency in Cloud Computing: A Systematic Literature Review of Definitions and Metrics', in *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*, 2015, pp. 83–92.
- [73] M. Becker, S. Lehrig, and S. Becker, 'Systematically deriving quality metrics for cloud computing systems', in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 2015, pp. 169–174.
- [74] G. Aceto, A. Botta, W. De Donato, and A. Pescapè, 'Cloud monitoring: A survey', *Comput. Netw.*, vol. 57, no. 9, pp. 2093–2115, 2013.
- [75] O. Moser, F. Rosenberg, and S. Dustdar, 'Non-intrusive monitoring and service adaptation for WS-BPEL', in *Proceedings of the 17th international conference on World Wide Web*, 2008, pp. 815–824.
- [76] K. Mahbub and G. Spanoudakis, 'Monitoring WS-Agreements: An Event Calculus-Based Approach', in *Test and Analysis of Web Services*, Springer, 2007, pp. 265–306.
- [77] F. Curbera, M. J. Duffler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, 'Colombo: Lightweight middleware for service-oriented computing', *IBM Syst. J.*, vol. 44, no. 4, pp. 799–820, 2005.
- [78] J. Povedano-Molina, J. M. Lopez-Vega, J. M. Lopez-Soler, A. Corradi, and L. Foschini, 'DARGOS: A highly adaptable and scalable monitoring architecture for multi-tenant Clouds', *Future Gener. Comput. Syst.*, vol. 29, no. 8, pp. 2041–2056, 2013.
- [79] D. Chaves, S. Aparecida, R. B. Uriarte, and C. B. Westphall, 'Toward an architecture for monitoring private clouds', *Commun. Mag. IEEE*, vol. 49, no. 12, pp. 130–137, 2011.

- [80] B. Konig, J. M. Alcaraz Calero, and J. Kirschnick, 'Elastic monitoring framework for cloud infrastructures', *Commun. IET*, vol. 6, no. 10, pp. 1306–1315, 2012.
- [81] I. Konstantinou, E. Angelou, C. Boumpouka, D. Tsoumakos, and N. Koziris, 'On the elasticity of nosql databases over cloud management platforms', in *Proceedings of the 20th ACM international conference on Information and knowledge management*, 2011, pp. 2385–2388.
- [82] A. Brinkmann, C. Fiehe, A. Litvina, I. Luck, L. Nagel, K. Narayanan, F. Ostermair, and W. Thronicke, 'Scalable monitoring system for clouds', in *Utility and Cloud Computing (UCC), 2013 IEEE/ACM 6th International Conference on*, 2013, pp. 351–356.
- [83] T. Goldschmidt, A. Jansen, H. Koziol, J. Doppelhamer, and H. P. Breivold, 'Scalability and Robustness of Time-Series Databases for Cloud-Native Monitoring of Industrial Processes', in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, 2014, pp. 602–609.
- [84] L. Larsson, D. Henriksson, and E. Elmroth, 'Scheduling and monitoring of internally structured services in Cloud federations', in *Computers and Communications (ISCC), 2011 IEEE Symposium on*, 2011, pp. 173–178.
- [85] V. C. Emeakaroha, T. C. Ferreto, M. A. Netto, I. Brandic, and C. A. De Rose, 'Casvid: Application level monitoring for sla violation detection in clouds', in *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, 2012, pp. 499–508.
- [86] H. Kung, C.-K. Lin, and D. Mah, 'CloudSense: Continuous Fine-Grain Cloud Monitoring with Compressive Sensing.', in *HotCloud*, 2011.
- [87] S. Meng, L. Liu, and T. Wang, 'State monitoring in cloud datacenters', *Knowl. Data Eng. IEEE Trans. On*, vol. 23, no. 9, pp. 1328–1344, 2011.
- [88] P. Leitner, C. Inzinger, W. Hummer, B. Satzger, and S. Dustdar, 'Application-level performance monitoring of cloud services based on the complex event processing paradigm', in *2012 Fifth IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, 2012, pp. 1–8.
- [89] J. Shao, H. Wei, Q. Wang, and H. Mei, 'A runtime model based monitoring approach for cloud', in *2010 IEEE 3rd International Conference on Cloud Computing*, 2010, pp. 313–320.
- [90] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, 'Benchmarking cloud serving systems with YCSB', in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.
- [91] R. Cattell, 'Scalable SQL and NoSQL data stores', *ACM SIGMOD Rec.*, vol. 39, no. 4, pp. 12–27, 2011.
- [92] T. W. Wlodarczyk, 'Overview of time series storage and processing in a cloud environment', in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, 2012, pp. 625–628.
- [93] P. J. Sadalage and M. Fowler, *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2012.
- [94] A. Lakshman and P. Malik, 'Cassandra: a decentralized structured storage system', *ACM SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.
- [95] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, 'Benchmarking cloud serving systems with YCSB', in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.
- [96] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, 'PNUTS: Yahoo!'s hosted data serving platform', *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, 2008.
- [97] Y. Abubakar, T. Adeyi, and I. G. Auta, 'Performance evaluation of nosql systems using ycsb in a resource austere environment', *Perform. Eval.*, vol. 7, no. 8, 2014.
- [98] V. Abramova, J. Bernardino, and P. Furtado, 'Evaluating cassandra scalability with YCSB', in *Database and Expert Systems Applications*, 2014, pp. 199–207.
- [99] A. Gandini, M. Gribaudo, W. J. Knottenbelt, R. Osman, and P. Piazzolla, 'Performance evaluation of NoSQL databases', in *Computer Performance Engineering*, Springer, 2014, pp. 16–29.
- [100] J. Domaschka, D. Seybold, F. Griesinger, and D. Baur, 'Axe: A Novel Approach for Generic, Flexible, and Comprehensive Monitoring and Adaptation of Cross-Cloud Applications', in *European Conference on Service-Oriented and Cloud Computing*, 2015, pp. 184–196.
- [101] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, 'Freenet: A distributed anonymous information storage and retrieval system', in *Designing Privacy Enhancing Technologies*, 2001, pp. 46–66.
- [102] 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.
- [103] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut, 'Quality of service for workflows and web service processes', *Web Semant. Sci. Serv. Agents World Wide Web*, vol. 1, no. 3, pp. 281–308, 2004.

- [104] C. Cappiello, K. Kritikos, A. Metzger, M. Parkin, B. Pernici, P. Plebani, and M. Treiber, 'A quality model for service monitoring and adaptation', in *Workshop on Service Monitoring, Adaptation and Beyond*, 2009, p. 29.
- [105] L. Baresi, S. Guinea, and L. Pasquale, 'Self-healing BPEL processes with Dynamo and the JBoss rule engine', in *International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting*, 2007, pp. 11–20.
- [106] C. Inzinger, W. Hummer, B. Satzger, P. Leitner, and S. Dustdar, 'Generic event-based monitoring and adaptation methodology for heterogeneous distributed systems', *Softw. Pract. Exp.*, vol. 44, no. 7, pp. 805–822, 2014.
- [107] R. Popescu, A. Staikopoulos, A. Brogi, P. Liu, and S. Clarke, 'A formalized, taxonomy-driven approach to cross-layer application adaptation', *ACM Trans. Auton. Adapt. Syst. TAAS*, vol. 7, no. 1, p. 7, 2012.
- [108] S. Guinea, G. Kecskemeti, A. Marconi, and B. Wetzstein, 'Multi-layered monitoring and adaptation', in *Service-Oriented Computing*, Springer, 2011, pp. 359–373.
- [109] A. Zengin, R. Kazhamiakin, and M. Pistore, 'Clam: Cross-layer management of adaptation decisions for service-based applications', in *2011 IEEE International Conference on Web Services*, 2011, pp. 698–699.
- [110] O. Bračevac, S. Erdweg, G. Salvaneschi, and M. Mezini, 'CPL: a core language for cloud computing', in *Proceedings of the 15th International Conference on Modularity*, 2016, pp. 94–105.
- [111] T. Chen and R. Bahsoon, *Toward a Smarter Cloud: Self-Aware Autoscaling of Cloud Configurations and Resources*. IEEE COMPUTER SOC 10662 LOS VAQUEROS CIRCLE, PO BOX 3014, LOS ALAMITOS, CA 90720-1314 USA, 2015.
- [112] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, 'BPMN4TOSCA: A domain-specific language to model management plans for composite applications', in *International Workshop on Business Process Modeling Notation*, 2012, pp. 38–52.
- [113] T. Binz, G. Breiter, F. Leymann, and T. Spatzier, 'Portable Cloud Services Using TOSCA', *IEEE Internet Comput.*, vol. 16, no. 3, pp. 80–85, 2012.
- [114] T. Lorida-Botran, J. Miguel-Alonso, and J. A. Lozano, 'A review of auto-scaling techniques for elastic applications in cloud environments', *J. Grid Comput.*, vol. 12, no. 4, pp. 559–592, 2014.
- [115] Kritikos, K., Plexousakis, D. (2015). Multi-Cloud Application Design through Cloud Service Composition. In *Cloud*, 686-693.
- [116] Kritikos, K., Plexousakis, D. (2016). Subsumption Reasoning for QoS-based Matchmaking. In *ESOCC*.
- [117] Xiong, P., Pu, C., Zhu, X., Griffith, R. (2013). vperfguard: An automated model-driven framework for application performance diagnosis in consolidated cloud environments. In *ICPE*, 271–282.
- [118] Zeginis, C., Kritikos, K., Plexousakis, D. (2014). Event Pattern Discovery for Cross-Layer Adaptation of Multi-cloud Applications. In *ESOCC*, 138-147.

ANNEX A: LIST OF ABBREVIATIONS

List of abbreviation used into the document:

- API: Application Programming Interface
- BKM: Business Knowledge Model
- BPaaS: Business Process as a Service
- BPEL: Business Process Execution Language
- BPMN: Business Process Model and Notation
- CAMEL: Cloud Application Execution Modelling Language
- CAMP: Cloud Application Management for Applications
- CEP: Complex Event Processing
- CIMI: Cloud Infrastructure Management Interface
- COAPS API: Compatible One Application and Platform Service API
- DMN: Decision Model and Notation
- DT: Decision Table
- DSL: Domain Specific Language
- ETTTP: Estimated Time To Prototype
- IaaS: Infrastructure as a Service
- JVM: Java Virtual Machine
- OCL: Object Constraint Language
- OCC: Open Cloud Computing Interface
- OWL-Q: Web Ontology Language – Query Language
- QoS: Quality of Service
- PaaS: Platform as a Service
- RDBMS: Relational Database Management Systems
- REST: Representational State Transfer
- SaaS: Software as a Service
- SLA: Service Level Agreement
- SLO: Service Level Objective
- SOA: Service Oriented Architecture
- SOAP: Simple Object Access Protocol
- SRL: Scalability Rule Language
- TOSCA: Topology and Orchestration Specification for Cloud Applications
- TSDB: Time Series Database
- UML: Unified Modelling Language
- USDL: Unified Service Description Language

- VM: Virtual Machine
- WADL: Web Application Description Language
- WSDL: Web Service Description Language
- Web application ARchive
- YAML: YAML Ain't Markup Language
- YCSB: Yahoo Cloud Serving Benchmark