PaaSage

Model Based Cloud Platform Upperware

Deliverable D2.1.1e

CloudML Guide and Assessment Report (Extended)

Version: 1.0

D2.1.1e

Name, title and organisation of the scientific representative of the project's coordinator:

Mr Tom Williamson Tel: +33 4 9238 5072 Fax: +33 4 92385011 E-mail: tom.williamson@ercim.eu

Project website address: http://www.paasage.eu

Project	
Grant Agreement number	317715
Project acronym:	PaaSage
Project title:	Model Based Cloud Platform Upperware
Funding Scheme:	Integrated Project
Date of latest version of Annex I against which the assessment will be made:	29th August 2012
Document	
Period covered:	
Deliverable number:	D2.1.1e
Deliverable title	CloudML Guide and Assessment Report (Extended)
Contractual Date of Delivery:	30th September 2013 (M12)
Actual Date of Delivery:	25th November 2013
Editor (s):	Alessandro Rossini
Author (s):	Alessandro Rossini, Arnor Solberg, Daniel Romero, Jörg Domaschka, Kostas Magoutis, Nicolas Ferry, Tom Kirkham, Maciej Malawski, Bartosz Balis, Dariusz Krol, Achilleas Achilleos
Reviewer (s):	Philippe Massonet, Geir Horn
Participant(s):	Lutz Schubert, Keith Jeffery
Work package no.:	2
Work package title:	Languages
Work package leader:	Arnor Solberg
Distribution:	
Version/Revision:	1.0
Draft/Final:	Final
Total number of pages (including cover):	50

DISCLAIMER

PaaSage is a project funded in part by the European Union.

Contents

Executive Summary

Cloud computing provides a ubiquitous networked access to a shared and virtualised pool of computing capabilities that can be provisioned with minimal management effort. Cloud-based applications are applications that are deployed on cloud infrastructures and platforms, and delivered as services. PaaSage aims to facilitate the specification and execution of cloud-based applications by leveraging upon model-driven engineering (MDE) techniques and methods, and by exploiting multiple cloud infrastructures and platforms.

MDE is a branch of software engineering that aims at improving the productivity, quality, and cost-effectiveness of software development by shifting the paradigm from code-centric to model-centric. Models enable the abstraction from the implementation details of heterogeneous cloud services, while model transformations facilitate the automatic generation of the source code that exploits these services. This approach, which is commonly summarised as "model once, generate anywhere", is particularly relevant when it comes to the specification and execution of multi-cloud applications (*i.e.*, applications deployed across multiple clouds infrastructures and platforms), which allow exploiting the peculiarities of each cloud service and hence optimising performance, availability, and cost of the applications.

Models are frequently specified using domain-specific languages (DSLs), which are tailored to a specific domain of concern. In order to cover the necessary aspects of the specification and execution of multi-cloud applications, PaaSage encompasses a family of DSLs. These DSLs, namely CLOUDML, Saloon, WS-Agreement, and a language for scalability rules, allow for modelling multiple concerns of multi-cloud applications, such as provisioning, deployment, requirements, goals, SLAs, and execution.

In this deliverable, we present an initial guide and assessment of the DSLs adopted in PaaSage by:

- relating each DSL to the PaaSage life-cycle phases;
- describing the domain covered by each DSL;
- describing the capabilities offered by each DSL;
- discussing the integration of the DSLs;
- exemplifying the usage of the DSLs by means of a running example.

In particular, this deliverable extends D2.1.1 [33] with an initial assessment of how the DSLs adopted in PaaSage are suitable for the specification and execution of the new use cases of the PaaSage Enlarged project.

Please note that these DSLs are under development and will evolve throughout the course of the PaaSage project. Hence, the capabilities offered by these DSLs and presented in this deliverable reflect our understanding of the requirements of PaaSage at month 12.

Please also note that a technical specification of these DSLs is beyond the scope of this deliverable. Low-level technical details, such as specifications of metamodels, grammars, and semantics, will be provided in D2.1.2 [32].

Intended Audience

This deliverable is a public document intended for readers with some experience with cloud computing and modelling, as well as some familiarity with the initial architecture design of PaaSage, as described in D1.6.1 [18].

For the external reader, this deliverable provides an insight of the domain covered- and the capabilities offered by each DSL, as well as a justification for their adoption in the context of the PaaSage life-cycle phases.

For the research partners in PaaSage, this deliverable provides an understanding of the elements of each DSL that can be manipulated by the components of the PaaSage platform.

For the industrial partners in PaaSage, this deliverable provides an understanding of the elements of each DSL that can be used for modelling the use cases of PaaSage.

1 Introduction

Nowadays, software systems are leveraging upon an aggregation of dedicated solutions, which leads to the design of large-scale, distributed, dynamic systems. However, the complexity of managing such systems challenges current software engineering approaches.

Dynamically adaptive systems (DAS) [21] have recently emerged to cope with this challenge by enabling the continuous design and adaptation of complex software systems. DAS facilitates handling short-term changes in the execution environment as well as long-term changes in the system requirements [24]. However, the focus of DAS is typically limited to the application itself rather than the underlying infrastructure and platform.

Cloud computing provides a ubiquitous networked access to a shared and virtualised pool of computing capabilities (*e.g.*, network, storage, processing, and memory) that can be provisioned with minimal management effort [23]. In contrast to DAS, cloud computing enables the management of the complete software stack, *i.e.*, infrastructure, platform, and application, where each layer is exposed as a service. In particular, cloud computing offers two types of scalability: vertical (*e.g.*, increase or decrease the virtual resources allocated to a virtual machine) and horizontal (*e.g.*, increase or decrease the number of virtual machines).

A key challenge is then to enable dynamically adaptive cloud-based applications, *i.e.*, to integrate the management capabilities of recent cloud solutions with software engineering approaches, techniques, and methods of DAS. PaaSage aims to tackle this challenge by leveraging upon model-driven engineering (MDE) techniques and methods, and by exploiting multiple cloud infrastructures and platforms.

MDE is a branch of software engineering that aims at improving the productivity, quality, and cost-effectiveness of software development by shifting the paradigm from code-centric to model-centric. MDE promotes the use of models and model transformations as the primary assets in software development, where they are used to specify, simulate, generate, and manage software systems. This approach is particularly relevant when it comes to the specification and execution of multi-cloud applications (*i.e.*, applications deployed across multiple private, public, or hybrid cloud infrastructures and platforms), which allow exploiting the peculiarities of each cloud service and hence optimising performance, availability, and cost of the applications.

Models can be specified using general-purpose languages like the Unified Modeling Language (UML). However, to fully unfold the potential of MDE, models are frequently specified using domain-specific languages (DSLs), which are tailored to a specific domain of concern. In order to cover the necessary aspects of the specification and execution of multi-cloud applications, PaaSage encompasses a family of DSLs called Cloud Application Modelling and Execution Language (CAMEL). In particular, PaaSage provides the Cloud Modelling Language (CLOUDML), a DSL for modelling and enacting the provisioning and deployment of multi-cloud applications [14, 15]. Besides CLOUDML, PaaSage adopts: Saloon [31, 30, 29], a framework for specifying requirements and goals of multi-cloud applications, and selecting compatible cloud providers; WS-Agreement [1], a framework for creating SLAs and monitoring them at runtime; and a language for specifying scalability rules (currently subject to an ongoing assessment).

In this deliverable, we present an initial guide and assessment of the DSLs adopted in PaaSage by:

- relating each DSL to the PaaSage life-cycle phases;
- describing the domain covered by each DSL;
- describing the capabilities offered by each DSL;
- discussing the integration of the DSLs;
- exemplifying the usage of the DSLs by means of a running example.

In particular, this deliverable extends D2.1.1 [33] with an initial assessment of how the DSLs adopted in PaaSage are suitable for the specification and execution of the new use cases of the PaaSage Enlarged project (see Section 10).

Please note that these DSLs are under development and will evolve throughout the course of the PaaSage project. Hence, the capabilities offered by these DSLs and presented in this deliverable reflect our understanding of the requirements of PaaSage at month 12.

Please also note that a technical specification of these DSLs is beyond the scope of this deliverable. Low-level technical details, such as specifications of metamodels, grammars, and semantics, will be provided in D2.1.2 [32].

1.1 Structure of the document

The remainder of the document is organised as follows. Section 2 summarises the life-cycle of multi-cloud applications envisioned in PaaSage. Section 3 presents a running example based on generic architectural scenarios for PaaSage that will be adopted throughout the document. Sections 4, 5, 6, 7, and 8 present the DSLs adopted in PaaSage, and exemplify how they can be used for specifying and executing the running example. Section 9 compares the proposed approach with related work. Section 10 provides an initial assessment of how the DSLs adopted in PaaSage are suitable for the specification and execution of large-scale scientific workflow applications, large-scale simulations, and financial auditing applications. Finally, Section 11 draws some conclusions and outlines some plans for future work.

2 PaaSage life-cycle

PaaSage's model-driven methodology is based upon the key cloud life-cycle phases of configuration, deployment, and execution of multi-cloud applications (see Figure 1, *cf.* D1.6.1 [18] for more information about the PaaSage architecture).

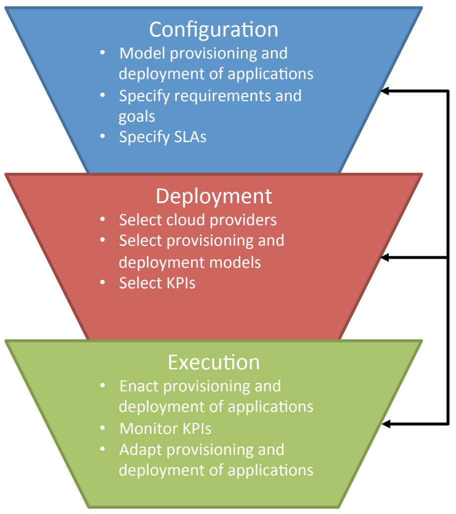

Configuration
Model provisioning and deployment of applications
Specify requirements and goals
Specify SLAs
Deployment
Select cloud providers
Select provisioning and deployment models
Select KPIs
Execution
Enact provisioning and deployment of applications
Monitor KPIs
Adapt provisioning and deployment of applications

Figure 1: The PaaSage life-cycle phases

The configuration phase is concerned with modelling the deployment of applications, specifying requirements and goals, and specifying SLAs. The deployment phase is concerned with matching the configuration models of applications with the profile of cloud providers, and selecting the most suitable deployment models based on requirements and goals, SLAs, and historical data about the executions of applications. Finally, the execution phase is concerned with monitoring and recording key performance indicators (KPIs), and managing the execution of applications accordingly.

In order to facilitate the integration across the components responsible for each life-cycle phase, PaaSage adopts a series of interlinked models, which are progressively refined throughout the PaaSage work-flow (see Figure 2).

Configuration: The users of PaaSage use CLOUDML, Saloon, and WS-Agreement to design a *configuration model*, which specifies the deployment

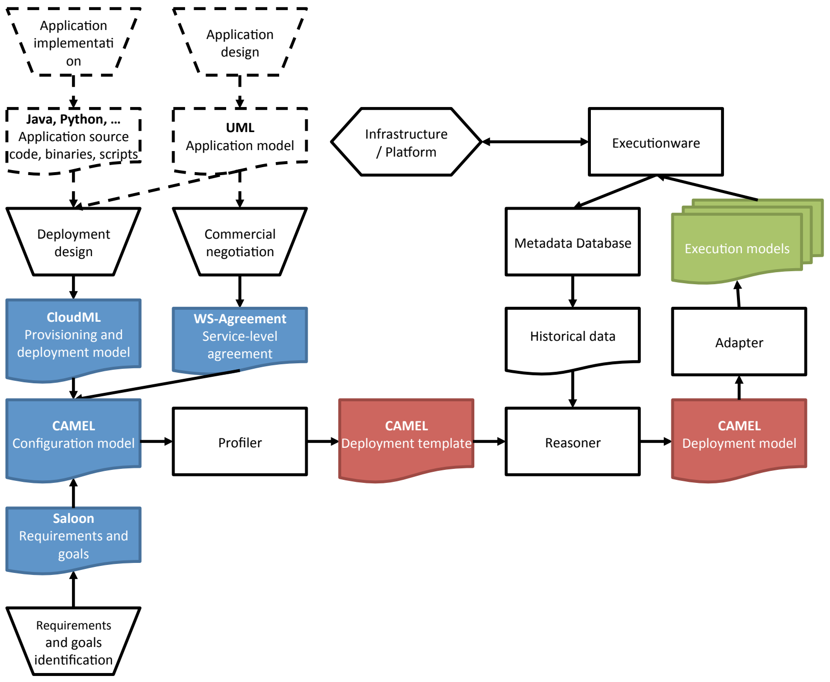

Application implementati on
Application design
Java, Python, … Application source code, binaries, scripts
UML Application model
Infrastructure / Platform
Executionware
Deployment design
Commercial negotiation
Metadata Database
Execution models
CloudML Provisioning and deployment model
WS-Agreement Service-level agreement
Historical data
Adapter
CAMEL Configuration model
Profiler
CAMEL Deployment template
Reasoner
CAMEL Deployment model
Saloon Requirements and goals
Requirements and goals identification

Figure 2: The PaaSage work-flow

of applications along with their requirements and goals in a cloud provider-independent way. This allows the users of PaaSage to express their requirements and goals at the level of abstraction that is suitable for their technical and business needs.

Deployment: The Profiler component (*cf.* D1.6.1 [18]) consumes the configuration model, matches this model with the profile of cloud providers, and produces a *deployment template*, which specifies the deployment of applications along with their requirements, goals, and compatible cloud providers. The Reasoner component (*cf.* D1.6.1 [18]) consumes the deployment template from the Profiler and produces a *deployment model*, which specifies the deployment of applications along with their requirements and goals in a cloud provider-specific way. This model and historical data related to its execution are stored in the Metadata Database component (*cf.* D1.6.1 [18]), which allows the Reasoner to look at the performance of previous deployment models when producing a new one. This model can also be shared and reused across different PaaSage platforms, which enables a knowledge foundation that related services such as the Social Network (*cf.* D1.6.1 [18]) will be built upon.

Execution: The Adapter component (*cf.* D1.6.1 [18]) consumes the deployment model from the Reasoner and produces *execution models*, which complement the deployment model with deployment scripts along with monitoring and scalability rules that are necessary for managing the execution. This allows for the dynamic adaptation of multi-cloud applications and the maintenance of their quality of service (QoS) throughout their execution. Finally, the Executionware (*cf.* D1.6.1 [18]) consumes the execution models from the Adapter and enacts the deployment of the application components along with suitable infrastructure and platform services to support the execution.

In the following, we present a running example that will be adopted throughout the document to describe the DSLs.

3 SENSAPP running example

SENSAPP[1] is an open-source, service-oriented application for storing and exploiting large data sets collected from sensors and devices. It is designed to seamlessly bridge the gap between the Internet of things (IoT) and the cloud [25]. The SENSAPP application can register sensors, store their data, and notify clients when new data are pushed.

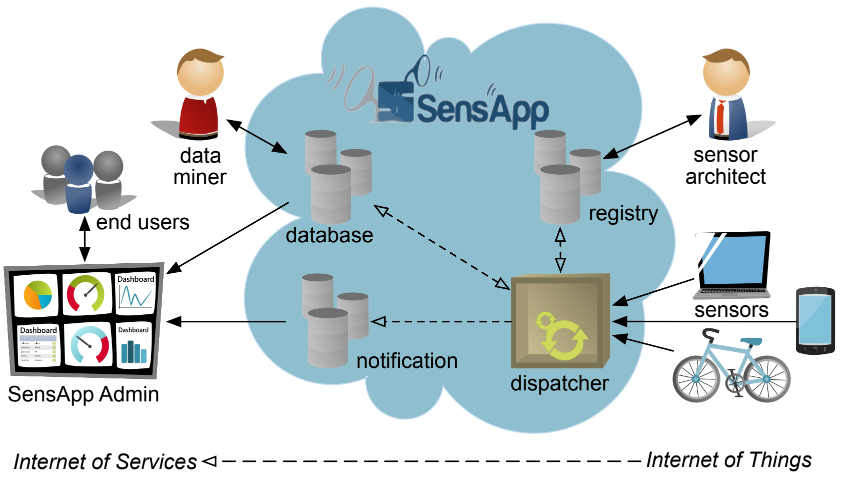


Figure 3: The SENSAPP architecture [25]

SENSAPP provides four essential components to support the definition of IoT applications (see Figure 3). The *Registry* component stores metadata about the sensors (*e.g.*, description and creation date). The *Database* component stores raw data from the sensors using a MongoDB database. The *Notifier* component sends notifications to third-party applications when relevant data are pushed (*e.g.*, when new data collected by air quality sensors become available). The *Dispatcher* component orchestrates the other components: it receives data from

[1] http://sensapp.org

the sensors, stores these data in the Database according to the metadata from the Registry, and then triggers the notification mechanisms for the new data. Finally, the *Admin* component provides capabilities to manage sensors and visualise data using a graphical user interface.

In the following, we adopt a running example illustrating different scenarios of provisioning and deployment of SENSAPP. First, the Dispatcher and the Database are deployed on a private cloud, the Notifier on a public cloud, and the Admin on another public cloud. Then, the virtual machines are scaled vertically while the Dispatcher, the Database, and the Notifier are scaled horizontally to tackle with the increase of requests. This running example can be regarded as an instance of the generic architectural scenarios for PaaSage presented in D1.6.1 [18], which are intended to cover the requirements of the PaaSage use cases. In the following, we show how the various aspect of the running example can be specified using the DSLs adopted in PaaSage.

4 Provisioning and deployment: CLOUDML

Provisioning and deployment models specify the topology of virtual machines and application components of cloud-based applications. PaaSage envisions the usage of provisioning and deployment models in all life-cycle phases of configuration, deployment, and execution (see Section 2), meaning that they are progressively refined throughout the PaaSage work-flow. For this purpose, we adopt CLOUDML[2] [14, 15]. CLOUDML consists of a tool-supported DSL for modelling and enacting the provisioning and deployment of multi-cloud applications as well as for facilitating their dynamic adaptation by leveraging upon MDE techniques and methods.

The life-cycle phases of configuration, deployment, and execution motivate for the following requirements for CLOUDML:

Cloud provider-independence (R_1): CLOUDML should support a cloud provider-agnostic specification of the provisioning and deployment, which will simplify the design of multi-cloud applications and prevent vendor lock-in;

Reusability (R_2): CLOUDML should support the specification of types of components that can be seamlessly reused, which will ease the modelling of the provisioning and deployment;

Modularity (R_3): CLOUDML should support the specification of modular, loosely-coupled components that can be seamlessly substituted, which will facilitate the dynamic adaptation of the provisioning and deployment;

[2] `http://cloumdml.org`

Abstraction (R_4): CLOUDML should provide an up-to-date, abstract representation of the running system, which will facilitate reasoning, simulation, and validation of adaptation actions before their actual executions.

4.1 Cloud provider-independent and -specific models

CLOUDML allows the specification of provisioning and deployment at various levels of abstraction. The two proposed levels are:

- the Cloud Provider-Independent Model (CPIM), which specifies the provisioning and deployment of a multi-cloud application in a cloud provider-independent way;
- the Cloud Provider-Specific Model (CPSM), which refines the CPIM and specifies the provisioning and deployment of a multi-cloud application in a cloud provider-specific way.

This two-level approach is agnostic to any development paradigm and technology, meaning that the application developers can design and implement their applications based on their preferred paradigms and technologies.

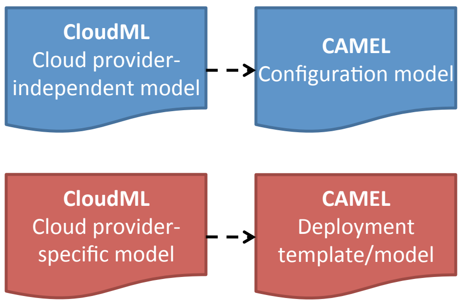

CloudML
Cloud provider-independent model
CAMEL
Configuration model
CloudML
Cloud provider-specific model
CAMEL
Deployment template/model

Figure 4: The CLOUDML modelling stack

Please note that the CPIM and CPSM constitute a subset of the configuration model and deployment template/model used in PaaSage, respectively (see Figures 4 and 2).

CLOUDML is inspired by component-based approaches and implements the *type-instance* pattern [3]. It allows expressing the following concepts (see Figure 5): *clouds*, *virtual machines*, *application components*, *ports*, and *relationships*.

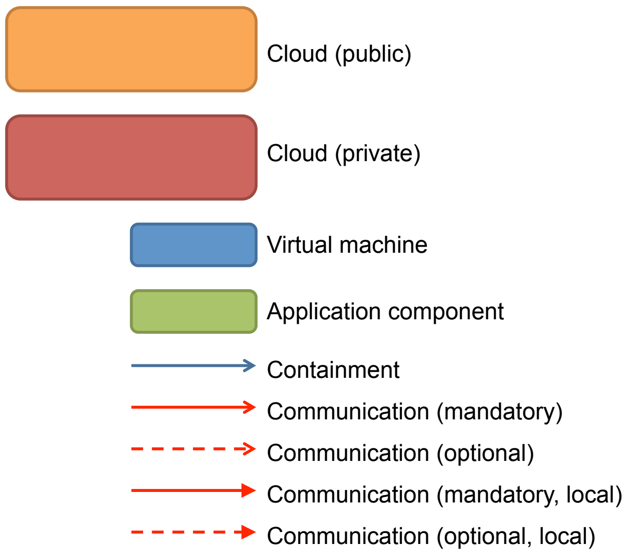


Figure 5: The CLOUDML visual syntax

Clouds

A *cloud* represents a collection of virtual machines on a particular cloud provider. This element can be parametrised by provisioning requirements (*e.g.*, location = Europe).

Virtual machines

A *virtual machine type* represents a reusable type of virtual machine. This element can also be parametrised by provisioning requirements (*e.g.*, 2 cores $\leq$ compute $\leq$ 4 cores, 2 GiB $\leq$ memory $\leq$ 4 GiB, storage $\geq$ 10 GiB) and operating system (*e.g.*, GNU/Linux or Windows) an. A *virtual machine instance* represents an instance of a virtual machine (*e.g.*, an instance of a virtual machine running GNU/Linux).

Application components

An *application component type* represents a reusable type of application component. In particular, an application component type can be *internal*, meaning that it represents a component to be deployed on a virtual machine (*e.g.*, a servlet, a Tomcat servlet container, and a MongoDB database), or *external*, meaning that it represents a component managed and provided as a service by an external party (*e.g.*, the Google Maps service). An application component type can be associated to *resources* specifying how to manage their deployment life-cycle (*e.g.*, download the servlet from `http://cloudml.org/`, install it, and start it). An *application component instance* represents an instance of an application component (*e.g.*, an instance of Tomcat deployed on an instance of a virtual machine running GNU/Linux).

Ports

A *port* represents the interface of a feature of an application component. In particular, a port can be *provided*, meaning that it serves a feature provided by the application component (*e.g.*, Tomcat provides a Java servlet container), or *required*, meaning that it consumes a feature required by the application component (*e.g.*, the servlet requires a Java servlet container). A required port can be *mandatory*, meaning that the application component requesting the feature depends on the application component providing the feature (*e.g.*, the servlet depends on MongoDB and so MongoDB has to be deployed before the servlet). A required port can also be *local*, meaning that the application component requesting the feature and the application component providing the feature have to be deployed on the same virtual machine (*e.g.*, the servlet and MongoDB have to be deployed on the same virtual machine).

Relationships

Finally, a *relationship* represents a relationship between *ports* of two application components. In particular, a relationship can be a *communication*, meaning that an application component communicates with another application component through an (IP-based) communication channel (*e.g.*, a servlet communicates with another servlet through Hypertext Transfer Protocol (HTTP) on port 8080), or a *containment*, meaning that an application component is contained by another application component (*e.g.*, the servlet is contained by Tomcat). A relationship type can be associated to *resources* specifying how to configure the application component types in order to communicate with- or contain each other.

4.2 Execution

As mentioned, the CPSM specifies the components of a multi-cloud application of and, in addition, which components shall be deployed on which cloud environments using which credentials and configurations. In order to bring the application to life, the content of the CPSM has to be transformed into operations executed against the interface of the actual cloud providers the components shall be deployed on. This is performed by the Executionware.

The Adapter splits the content of the CPSM on a per-cloud environment basis and passes each chunk to a Deployer instance in the Executionware. Each Deployer is responsible for a specific cloud environment. First, it replaces general configuration parameters to a set of specific parameters understandable by the cloud environment. For instance, the Amazon EC2 Deployer may translate a virtual machine running GNU/Linux and parametrised by 4 compute cores and 16 GiB of memory to an *m3.xlarge*[3] virtual machine with an *Ubuntu Sec* Amazon Machine Image (AMI), and take similar steps for the storage and networking services. Finally, the Deployers instantiate the application components.

In the following, we build upon the SENSAPP running example (see Section 3) to exemplify the usage of CLOUDML.

4.3 Running example

During the life-cycle phase of configuration, the users of PaaSage use the Modeller to design a CPIM of SENSAPP. This CPIM describes a possible provisioning and deployment of SENSAPP in a cloud provider-independent way. Figure 6 shows a CPIM of SENSAPP using the CLOUDML visual syntax (see Figure 5).

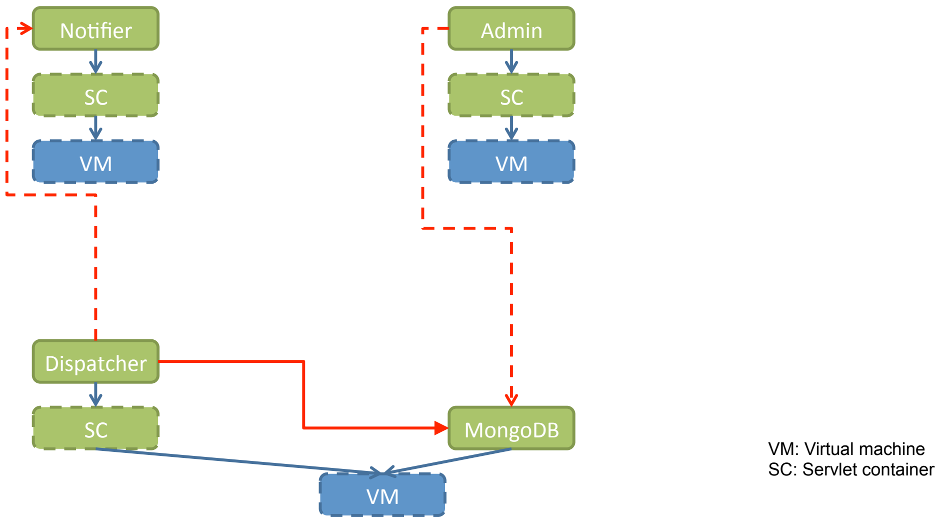


Figure 6: A sample CPIM for SENSAPP

[3]http://aws.amazon.com/ec2/instance-types/

The CPIM consists of a Dispatcher, a Notifier, and an Admin servlet from SensApp. These servlets are contained by generic servlet containers, which in turn are contained by generic virtual machines. The Dispatcher communicates with MongoDB. This connection is mandatory, meaning that the Dispatcher depends on MongoDB and so MongoDB has to be deployed before the Dispatcher (*i.e.*, deploy MongoDB, deploy the Dispatcher, and configure the connection from the Dispatcher to MongoDB). This connection is also local, meaning that the Dispatcher and MongoDB have to be contained by the same virtual machine. The Dispatcher also communicates with the Notifier. This connection is optional, meaning that the Dispatcher does not depend on the Notifier. Finally, the Admin communicates with MongoDB. This connection is optional.

Please note that the CPIM allows the specifications of requirements at different levels of abstractions. The generic servlet containers and virtual machines denote that SensApp can be executed on any servlet container (*e.g.*, Tomcat or Jetty, etc.) on any virtual machine (*e.g.*, running Ubuntu Linux or Windows Server, and having 2 or 16 compute cores). In contrast, MongoDB denotes that SensApp can only be executed with this database.

During the life-cycle phase of deployment, the CPIM is refined into a CPSM of SensApp. This CPSM specifies the provisioning and deployment in a cloud provider-specific way. The refinement of a CPIM into a CPSM can be performed either manually or automatically. In the former case, the users of PaaSage manually select the cloud providers and the cloud provider-specific aspects of the provisioning and deployment through the Modeller. In the latter case, the users of PaaSage let the Profiler match the requirements and goals with the compatible providers (see Section 5), and then let the Reasoner select the most suitable CPSM based on requirements and goals, SLAs, and historical data about the executions of applications (*cf.* D1.6.1 [18]).

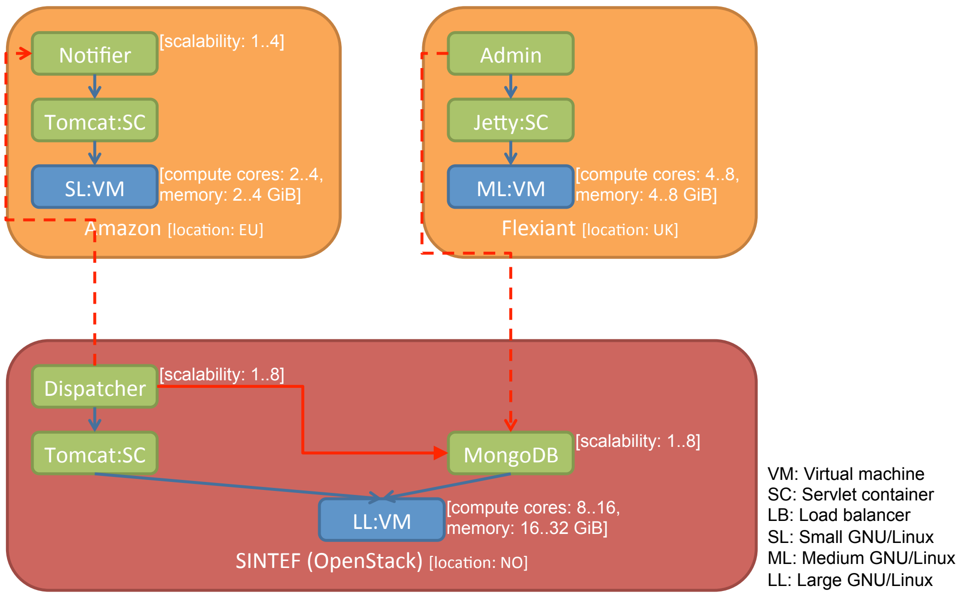


Figure 7: A sample CPSM for SENSAPP

Figure 7 shows a CPSM of SENSAPP. The generic virtual machines are refined to a Small Linux (SL), a Medium Linux (ML), and a Large Linux (LL) virtual machine. The SL virtual machine has 2 to 4 compute cores and 2 to 4 GiB memory, meaning that the Execution Engine in the Executionware can scale vertically these resources within these ranges. Similarly, the ML virtual machine has 4 to 8 compute cores and 4 to 8 GiB memory, while the LL virtual machine has 8 to 16 compute cores and 16 to 32 GiB memory. The generic servlet containers are refined to two Tomcat and a Jetty. The Dispatcher and MongoDB are refined to be scalable from from 1 to 8 times, meaning that the Execution Engine can scale horizontally their instances within this range. Similarly, the Notifier is refined to be scalable from 1 to 4 times. Finally, these virtual machines are partitioned into three clouds, namely a public cloud located in the EU based on Amazon EC2[4], another public cloud located in the UK based on Flexiant[5], and a third private SINTEF cloud located in Norway and based on OpenStack[6].

During the life-cycle phase of execution, the Executionware consumes the CPSM and enacts the provisioning and deployment of the SENSAPP components along with suitable infrastructure and platform services to support the execution.

[4]`http://aws.amazon.com/ec2/`
[5]`http://www.flexiant.com/`
[6]`http://www.openstack.org/`

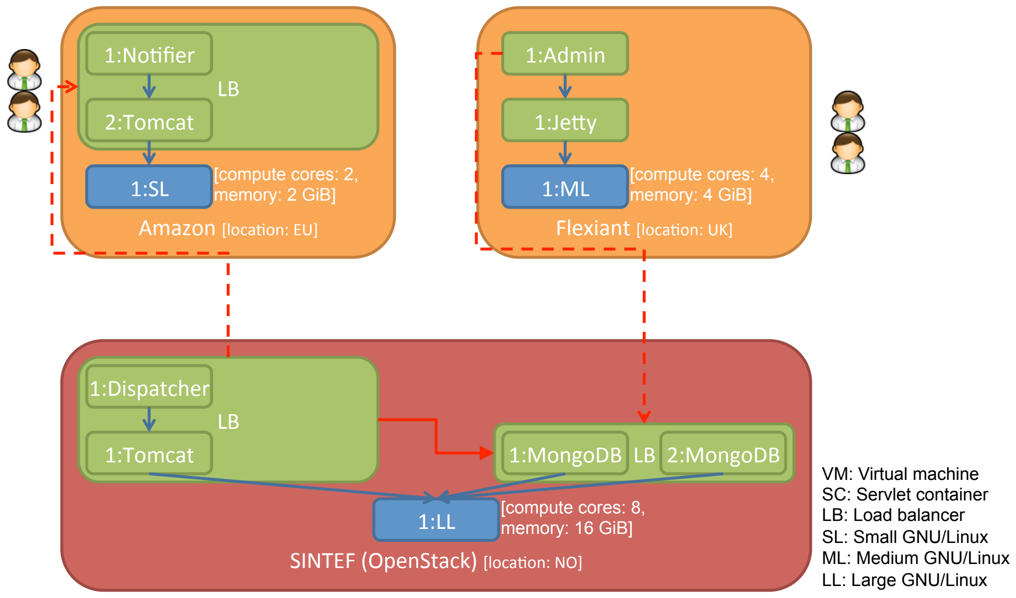


Figure 8: A first snapshot of the CPSM at run-time

Figure 8 shows a first snapshot of the CPSM at run-time. The Executionware has deployed an instance of the Notifier contained by an instance of Tomcat behind a Load Balancer on an instance of SL with 2 compute cores and 2 GiB memory. Similarly, it has deployed an instance of the Dispatcher contained by an instance of Tomcat behind a Load Balancer, as well as an instance of MongoDB behind another Load Balancer, on an instance of LL with 8 compute cores and 16 GiB memory.

Figure 9 shows a second snapshot of the CPSM at run-time. The Monitors in the Executionware have measured an increased number of clients. This has led the Execution Engine to apply scalability strategies such as *scale up* (*i.e.*, increase the size of virtual compute cores and virtual memory on a virtual machine) and *scale out* (*i.e.*, add more instances of application components) in response to the increase of requests (see Section 7). In particular, the Execution engine has scaled up LL by increasing its resources to 12 compute cores and 24 GiB memory. Similarly, the Execution Engine has scaled out the Dispatcher by deploying another instance of it (contained by another instance of Tomcat) on LL.

Please note that this is just one of the possible scalability strategies. For instance, another scalability strategy could scale out LL (instead of scale up) by deploying another instance of it. The selection of the most suitable scalability strategy is performed by the Reasoner based on requirements and goals, SLAs, and historical data about the executions of applications (*cf.* D1.6.1 [18]).

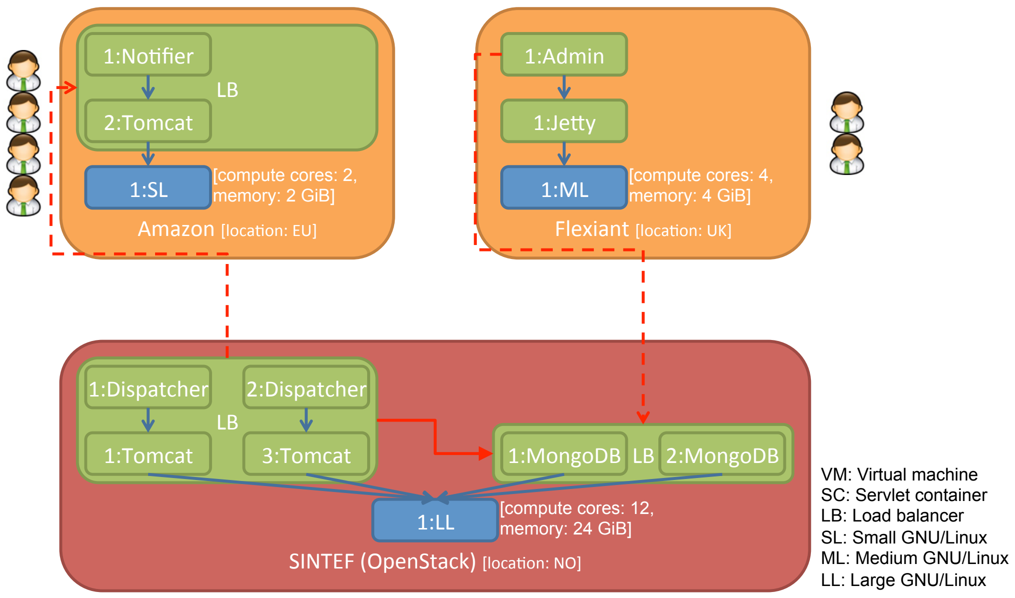


Figure 9: A second snapshot of the CPSM: The resources of LL are scaled up and the Dispatcher is scaled out in the SINTEF cloud

Finally, Figure 10 shows a last snapshot of the CPSM at run-time. The Execution Engine has scaled up SL by increasing its resources to 4 compute cores and 4 GiB memory. Moreover, the Execution Engine has scaled out the Notifier by deploying another instance of it (contained by another instance of Tomcat) on SL.

As illustrated through our SENSAPP use case, CLOUDML can be used to provision, deploy, and adapt multi-cloud applications. The following list summarises how it fulfills the requirements presented in Section 4.

Cloud provider-independence (R_1)**:** The layering of the CLOUDML modelling stack into CPIMs and CPSMs enables a cloud provider-independent modelling of cloud-based applications.

Reusability (R_2)**:** The type-instance pattern in CLOUDML ensures that the types of components can be seamlessly reused within multiple CPIMs and CPSMs.

Modularity (R_3)**:** The component-based design of CLOUDML ensures that the components of CPIMs and CPSMs can be seamlessly substituted.

Abstraction (R_4)**:** The usage of CPSMs at run-time provides an abstract and up-to-date representation of the running system that can be dynamically manipulated.

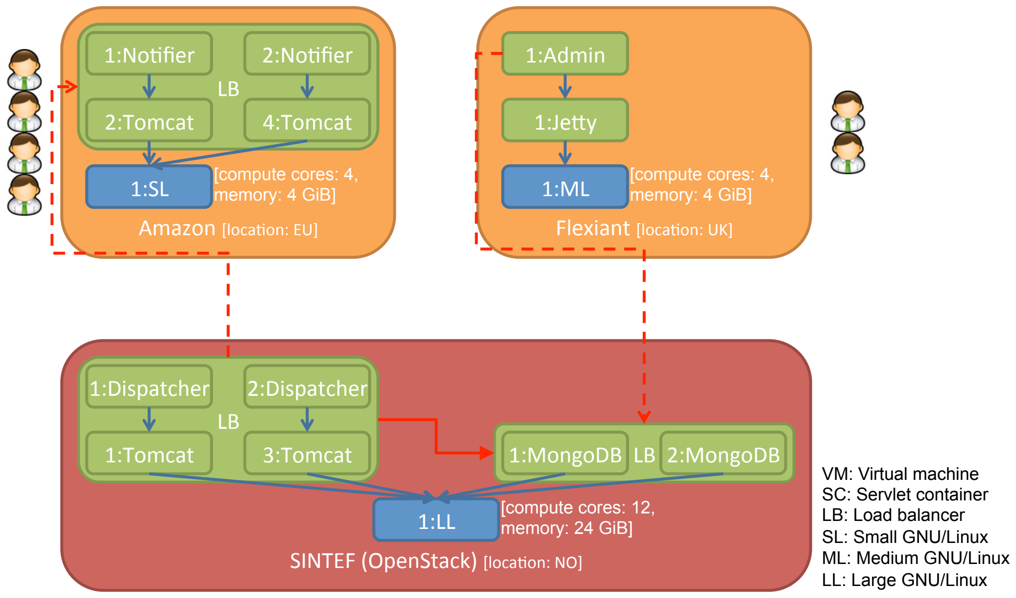


Figure 10: A last snapshot of the CPSM: The resources of SL are scaled up and the Notifier is scaled out in the Amazon cloud

CLOUDML is available as an open-source project[7]. It is implemented with Java and Scala as programming languages and Maven as a build tool. The current codebase consists of around 5 000 lines of Java code and 1 000 lines of Scala code. The CLOUDML models and metamodels are represented as plain Java objects. These models can be serialised in either JavaScript Object Notation (JSON) or XML Metadata Interchange (XMI). The JSON and XMI codecs are based on Kotlin[8] and the Kevoree Modeling Framework (KMF)[9] [16], respectively.

5 Cloud providers, requirements, and goals: Saloon

The ability to run and manage multi-cloud applications allows exploiting the peculiarities of different cloud providers and hence optimising performance, availability, and cost of the applications. However, the cloud providers are typically heterogeneous and the provided capabilities are often incompatible. This may reduce the set of compatible cloud providers when provisioning and deploying

[7]`https://github.com/SINTEF-9012/cloudml`
[8]`http://kotlin.jetbrains.org/`
[9]`https://github.com/dukeboard/kevoree-modeling-framework`

applications. PaaSage envisions the usage of profiles of cloud providers in the life-cycle phase of configuration and deployment (see Section 2) for matching the deployment models of applications with the compatible cloud providers. For this purpose, we adopt a modified version of Saloon [31, 30, 29]. Saloon is a framework for specifying requirements and goals of multi-cloud applications, and selecting compatible cloud providers, by leveraging upon feature models [6] and ontologies [17].

5.1 Variability of cloud providers

Feature models have been introduced as part of feature-oriented domain analysis (FODA) [20], and specify the commonality and variability of software systems, where the features are regarded as distinctive characteristics of a software system [20]. Feature models consist of tree-based structures, where the nodes represent the features and the edges represent the variability. Each feature can have multiple feature groups as children.

In traditional features models, feature groups can be of four different types: *mandatory*, specifying that all features in the feature group must be selected; *optional*, specifying that any features in the group can be selected; *alternative (xor)*, specifying that exactly one feature in the group must be selected; and *or*, specifying that at least one feature in the group must be selected. In feature models with cardinalities [11, 5], feature groups can also have a *cardinality* specifying the lower and upper bound of features to be selected. Feature models can also be attached *dependencies* between features. In Saloon, feature models with cardinalities capture the commonality and variability of cloud providers.

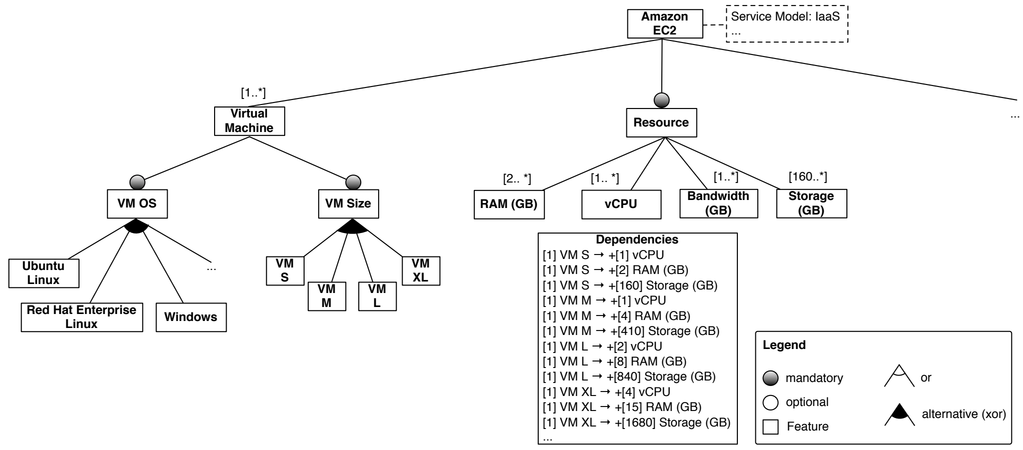


Figure 11: A sample Saloon feature model for Amazon EC2

Figure 11 shows a fragment of a feature model for Amazon EC2. According to this feature model, in Amazon EC2 we must select from one to many virtual machines. For each virtual machine, we must select the operating system (more precisely, the virtual machine image providing the operating system) and the size. The attached dependencies specify that virtual machines of specific sizes require specific resources; *e.g.*, a virtual machine of size S requires 1 compute core, 2 GiB memory, etc.

5.2 Requirements

Ontologies are formal and explicit specifications of a shared conceptualisation [10], which are used to describe the concepts and the relationships between these concepts. They aim at defining a vocabulary and knowledge representation of a concrete domain. In Saloon, the cloud ontology is used to deal with the syntactic and semantic heterogeneity to express the capabilities of different cloud providers.

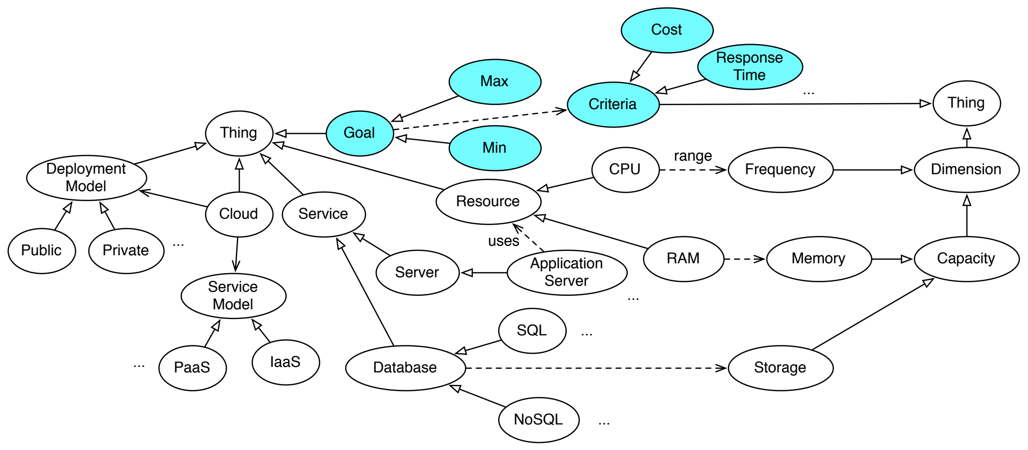


Figure 12: The Saloon cloud ontology

Figure 12 shows the cloud ontology of Saloon. This ontology contains technical elements and quantifiable elements. The former specifies the technical requirements supported by cloud providers, *e.g.*, application server and database. The latter specifies the dimensions associated to these technical requirements, *e.g.* compute core frequency or database size.

5.3 Goals

Saloon allows profiling cloud providers by means of feature models, and specifying application requirements by means of ontologies. However, the users of

PaaSage may not just want to choose a set of cloud providers that satisfy their application requirements. They may also want to specify some goals that have an important impact on their business, such as the minimisation of cost and/or application response time. In mathematics and computer science, this kind of goals are reified in an optimization problem composed by a real function which value has to be minimised or maximised [2]. In PaaSage, the cloud ontology is extended with such concepts (see blue concepts in Figure 12). In this way, this ontology enables the definition of minimisation and maximisation of user goals.

In the following, we build upon the SENSAPP running example (see Section 3) to exemplify the usage of Saloon.

5.4 Running example

During the life-cycle phase of configuration, the users of PaaSage use the Modeller to specify the requirements for each application component of SENSAPP. The Admin and the Notifier require a virtual machine with GNU/Linux on a public cloud. The Dispatcher also requires a virtual machine with GNU/Linux but on a private cloud. Figure 13 shows a fragment of the definition of requirements for the Admin and the mapping between the concepts in the ontology and the features in the feature models, which enable the identification of the compatible cloud providers. In the ontology, we select the concepts related to IaaS, public deployment model, and virtual machine with GNU/Linux. Among the feature models, we only consider five cloud providers: Amazon EC2, Flexiant, Rackspace[10], Google App Engine[11], and a private Cloud from SINTEF based on OpenStack.

During the life-cycle phase of deployment, the Profiler matches the requirements and goals of applications with the compatible profiles of cloud providers. Table 1 summarizes for each application component in SENSAPP the list of cloud providers that can be used for their deployment. The Admin and the Notifier component can be deployed on Amazon EC2, Rackspace, and Flexiant, which are public and IaaS providers. Google App Engine is excluded because it is a PaaS provider, while SINTEF (OpenStack) is excluded because it is a private provider. The Dispatcher component can only be deployed on SINTEF (OpenStack). This is the only candidate because Amazon EC2, Rackspace, and Flexiant are public, while Google App Engine is public and PaaS.

[10]http://www.rackspace.com/

[11]https://developers.google.com/appengine/

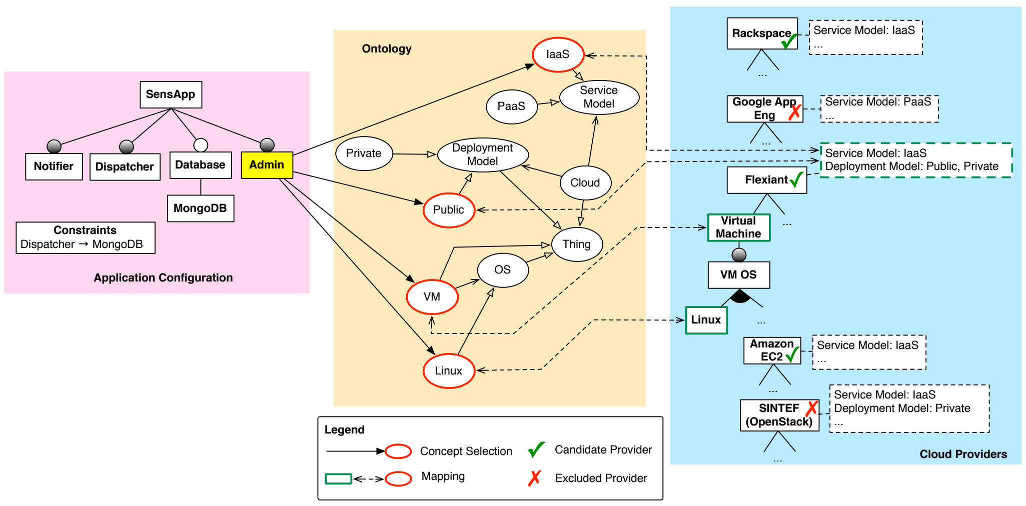


Figure 13: Sample Saloon models for SENSAPP (Excerpt)

Table 1: Candidate cloud provider for the deployment of each SENSAPP component

Appl. component	Provider	Candidate	Comments
Admin	Amazon EC2	✓	
	Flexiant	✓	
	Google App Engine	✗	PaaS
	Rackspace	✓	
	SINTEF (OpenStack)	✗	Private
Notifier	Amazon EC2	✓	
	Flexiant	✓	
	Google App Engine	✗	PaaS
	Rackspace	✓	
	SINTEF (OpenStack)	✗	Private
Dispatcher	Amazon EC2	✗	Public
	Flexiant	✗	Public
	Rackspace	✗	Public
	Google App Engine	✗	Public, PaaS
	SINTEF (OpenStack)	✓	

6 Service-level agreement: WS-Agreement

Service-level agreements (SLAs) support the business relationship between providers and consumers of services. In particular, a SLA between a service consumer and a service provider specifies one or more service-level objectives (SLOs), which are requirements of the service consumer and assurances by the service provider on the availability of resources and/or on service qualities. Since cloud infrastructure, platform, and applications can change dynamically, the management of SLAs should become flexible.

PaaSage envisions the usage of SLAs in the life-cycle phases of configuration and deployment (see Section 2) for finding the most suitable deployment model for multi-cloud applications to maintain the QoS throughout their execution. For this purpose, we adopt Web Services Agreement (WS-Agreement) standard [1] from the Open Grid Forum (OGF). WS-Agreement consists of a language and a protocol for advertising the capabilities of service providers, creating SLAs based on templates, and monitoring SLAs at run-time.

WS-Agreement extends the classical model of service discovery and usage by enabling service consumers not only to discover and use services, but also to dynamically negotiate the quality with which the service is provided [28]. This is mostly done automatically, and various approaches to the negotiation exists.

A popular Java-based implementation of WS-Agreement, namely WSAG4J[12], is under consideration for being adopted in PaaSage.

6.1 Running example

During the life-cycle phase of configuration, the users of PaaSage use the Modeller to specify the SLA for SENSAPP. Listings 1 shows a fragment of a WS-Agreement specifying response time requirements measured in milliseconds for the applications components of SENSAPP. The minimum and maximum response time for Notifier, Dispatcher, Admin, and MongoDB.

[12] `http://packcse0.scai.fraunhofer.de/wsag4j`

Listing 1: A WS-Agreement for SensApp

```
<wsag:CreationConstraints>
  <wsag:Item wsag:Name="Notifier">
    <wsag:ItemConstraint>
      <xs:element name="response_time_ms">
        <xs:simpleType>
          <xs:restriction base="xs:int">
            <xs:minInclusive value="1" />
            <xs:maxInclusive value="10" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </wsag:ItemConstraint>
  </wsag:Item>
  <wsag:Item wsag:Name="Admin">
    <wsag:ItemConstraint>
      <xs:element name="response_time_ms">
        <xs:simpleType>
          <xs:restriction base="xs:int">
            <xs:minInclusive value="1" />
            <xs:maxInclusive value="50" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </wsag:ItemConstraint>
  </wsag:Item>
  <wsag:Item wsag:Name="Dispatcher">
    <wsag:ItemConstraint>
      <xs:element name="response_time_ms">
        <xs:simpleType>
          <xs:restriction base="xs:int">
            <xs:minInclusive value="1" />
            <xs:maxInclusive value="5" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </wsag:ItemConstraint>
  </wsag:Item>
  <wsag:Item wsag:Name="MongoDB">
    <wsag:ItemConstraint>
      <xs:element name="response_time_ms">
        <xs:simpleType>
          <xs:restriction base="xs:int">
            <xs:minInclusive value="1" />
            <xs:maxInclusive value="5" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </wsag:ItemConstraint>
  </wsag:Item>
</wsag:CreationConstraints>
```

During the life-cycle phase of deployment, the Reasoner consumes the SLA and uses it along with the requirements and goals for finding the most suitable CPSM for SENSAPP to maintain the QoS of SENSAPP throughout its execution.

7 Scalability: Rules

Cloud providers typically offer virtual *computational* and *storage* resources [19]. In this respect, these resources are postulated to be infinite, and are typically sold on pay-per-use basis. The amount of provisioned resources will always constitute a trade-off between performance and cost, since their underprovisioning may compromise the former, while their overprovisioning may make the latter prohibitive. Hence, it is desirable that the applications are dynamically adapted so that the amount of used resources suits the changing demand.

A rudimentary approach would be to manually leave the adaptation to a human operator, such as a system administrator. This approach would have serious limitations. First, it would limit the reaction time to the operation hours of the human operator. Second, it would limit the quality of scaling to the experience of the human operator. Hence, in order to ensure that the applications are dynamically adapted to the changing demand, the adaptation should not be left to human operator alone.

Another, more sensible approach is to automate the adaptation of the applications by adopting a suitable technique, such as the automatic execution of scaling strategies. This approach maintains the QoS of the applications throughout their entire execution. PaaSage adopts scaling strategies in the form of scaling rules in the life-cycle phase of execution (see Section 2). For this purpose, in addition to the other DSLs, we will adopt a language for specifying scalability rules.

The Executionware consumes these scalability rules and maps them to properties offered by the selected cloud providers, if available. For instance, Amazon Web Services offers the *Auto Scaling* service[13] that scales applications up and down according to specified conditions. In case the selected cloud provider does not provide such a mechanism, or only an insufficient mechanism, the Executionware takes over this task. The specification of the rules is two-fold: the Reasoner can derive simple rules from the data in the Metadata Database, or select more complex rules provided by application developers and cloud experts through the Social Network.

The specification of a DSL for scalability rules is subject to Task T2.3 that will start at *month 13*. Hence, in the following, we only provide a high-level discussion of the specification of execution aspects in the context of PaaSage.

[13]`http://aws.amazon.com/autoscaling/`

7.1 Scope and actions

The scalability rules typically target virtual computing (*i.e.*, the size of compute core and memory of a virtual machine) and storage (*i.e.*, the size of virtual storage devices and databases) resources. In particular, the adaptation actions consist of four types: *scale up* (*i.e.*, increase the size of compute core and memory on a virtual machine or the size of a virtual storage device) and *scale out* (*i.e.*, add more virtual machines or more virtual storage devices), as well as the inverse adaptation actions *scale down* and *scale in*, respectively.

7.2 Triggers and emitters

The application of scalability rules is triggered by specific types of events (or sequence of events). These events may be emitted by different sources. In PaaSage, these events are emitted by two sources. The first source are the Monitors in the Executionware, which emit an event stream (*e.g.*, compute core load, memory consumption, storage consumption, and network traffic) for each application component and virtual machine. The second source is the Metadata Database, which contains all historical data about the execution of applications. Please note that due to the distributed nature of both multi-cloud applications and the PaaSage platform, the execution of adaptation actions may influence the system behaviour on a global- rather than local scale. Hence, the application of scalability rules can be considered as an event itself, which is recorded into the Metadata Database and emitted. This enables using the application of other scalability rules (or even themselves) as a trigger.

While the scalability rules derived by the Reasoner only take into account a single monitoring source and a single instance of an application component, those defined by application developers and cloud experts may be more complex and consider multiple monitoring sources and multiple instances of an application component. Hence, the scalability rules have to be expressive enough for specifying complex sequences of actions based on different events emitted by multiple sources. This is exactly the expressive power provided by existing event condition action (ECA) or event processing engines such as the open-source Esper engine[14]. A language from one of these engines (*e.g.*, the Esper Processing Language) is under consideration for being adopted in PaaSage.

[14]`http://esper.codehaus.org/`

7.3 Running example

In the following, we build upon the SENSAPP running example (see Section 3) to exemplify the possible management of scalability rules.

The Deployers consume the CPSM of SENSAPP (see Figure 7) and enact its provisioning and deployment. An instance of the Notifier and an instance of Tomcat are deployed on an instance of a Small Linux virtual machine on Amazon in the EU. An instance of the Dispatcher, an instance of Tomcat, and an instance of MongoDB are deployed on a Large Linux virtual machine of the private SINTEF (OpenStack) cloud in Norway. Finally, an instance of the Admin and an instance of Jetty are deployed on an instance of Medium Linux virtual machine on Flexiant in the UK.

The Deployers, Execution Engines, and Monitors also extract and process the scalability strategies from the CPSM of SENSAPP as well as from the general scalability recommendations with respect to the selected applications and cloud providers from the Metadata Database, as follows:

- Notifier: scales out to maximally 4 instances
- Dispatcher: scales out to maximally 8 instances
- MongoDB: scales out to maximally 8 instances

Please note that the up- and downscaling behaviour of the virtual machines depends on the selected cloud provider. In some cases, it can only be achieved by the complete reconfiguration and thus redeployment of the virtual machines. Since no further scalability strategies are specified by the application, the generic scalability rules provided by the cloud provider and/or external experts (*cf.* D1.6.1 [18]) are extracted, as follows:

- Scale out when the response time is below a specific threshold
- Scale in when the number of accesses is below a specific threshold
- Scale DB up when less than a specific number of rows are free
- Scale DB down when more than a specific number of rows are free

As mentioned, the corresponding scalability rules will be specified in an ECA-like format. At run-time, the Monitors will emit an event stream to match against the ECA rules given above to execute the corresponding adaptation actions.

8 Integration: Metadata Database

PaaSage envisions the usage a Metadata Database [27] for providing a holistic coverage of all information used in the life-cycle phases of configuration, deployment, and execution (see Section 2). In this respect, it stores the configuration, deployment, and execution models along with historical data about their execution. This approach is akin to models@run-time [24, 8], which enables the continuous evolution of multi-cloud applications with no strict boundaries between design-time and run-time activities. In the following, we discuss the mapping between these DSLs and the Metadata Database schema.

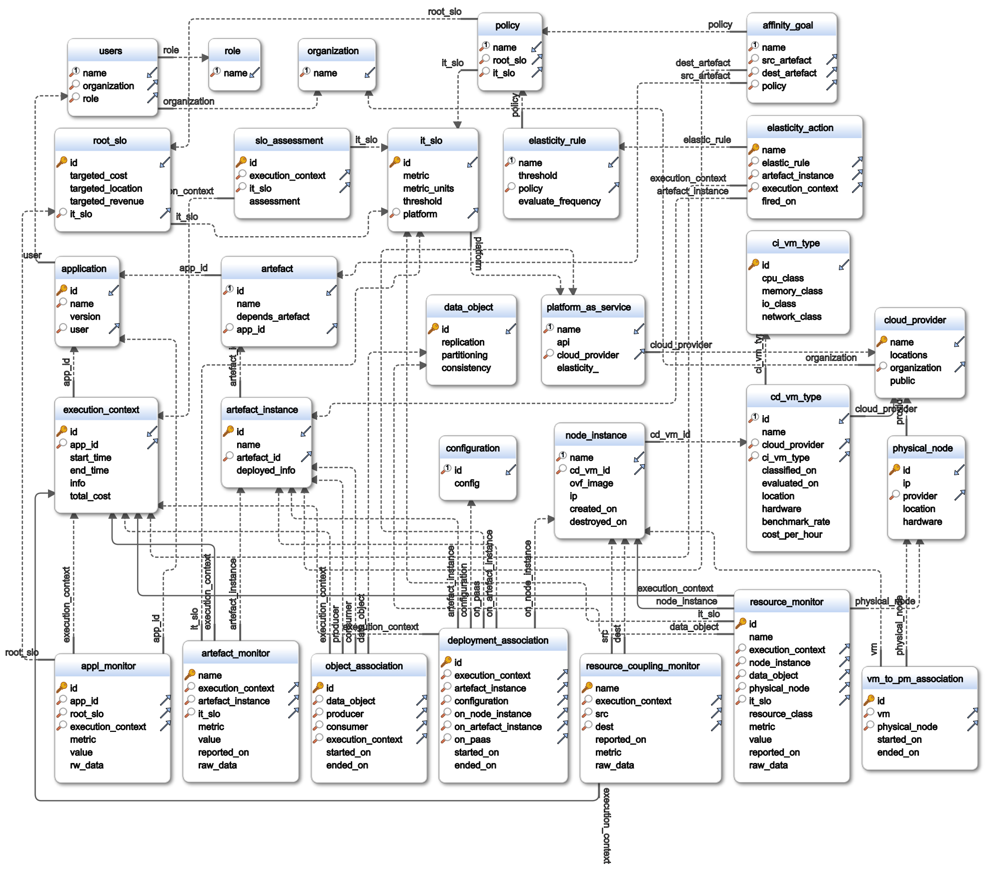


Figure 14: The Metadata Database schema

Figure 14 depicts the Metadata Database schema in traditional Entity-Relationship (ER) diagram notation. The Metadata Database schema consolidates the information captured by configuration, deployment, and execution models. Moreover, it extends these models with historical data about their execution such as monitoring data, event data, application of scalability rules, and adaptations. This information is consumed by the other components of the PaaSage platform to perform their analytics.

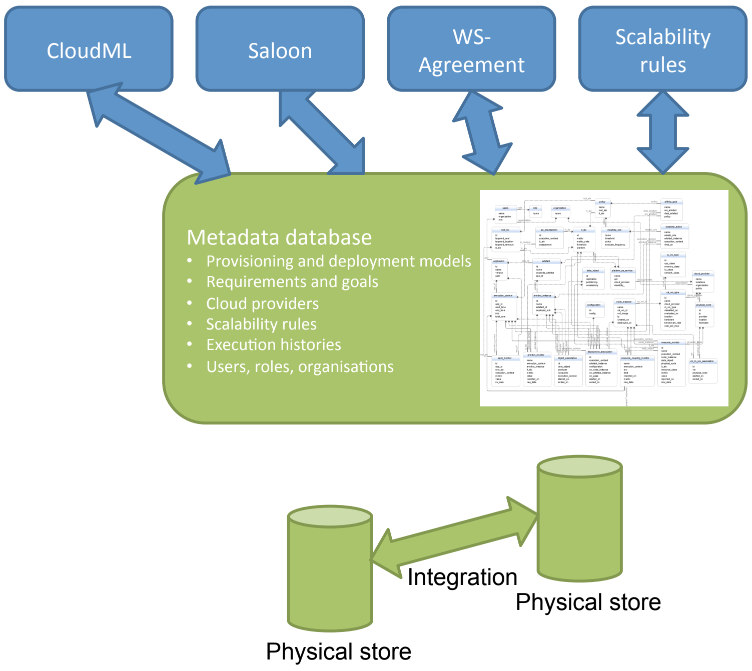


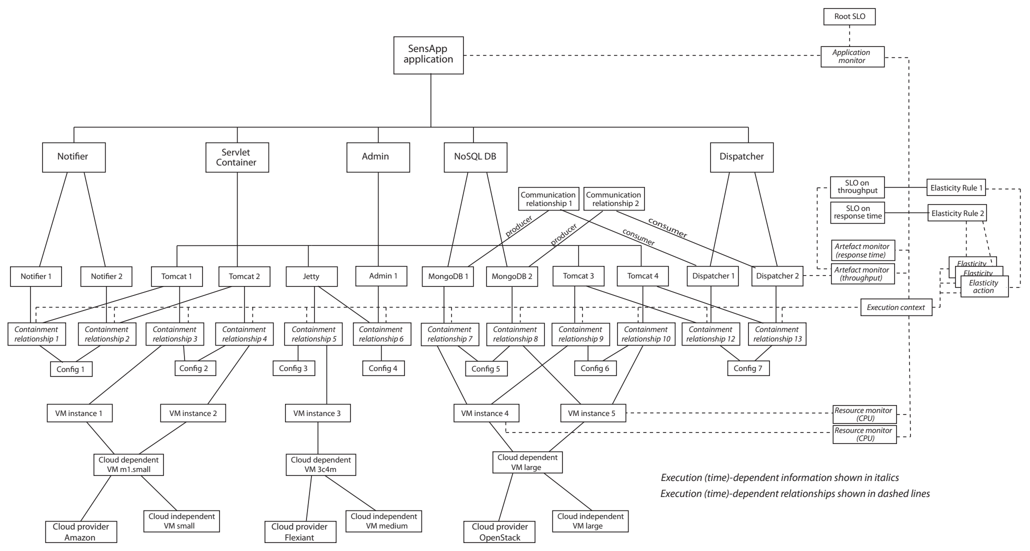
Figure 15: The mapping between DSLs and Metadata Database schema

Figure 15 depicts the relationship of the various models, each specified by means of a different DSL, with the Metadata Database schema. A key challenge is to ensure that such the mapping between the DSLs and the Metadata Database schema remains current and valid over time despite the independent evolution of the different DSLs adopted in PaaSage.

As mentioned, beyond consolidating the various models, the Metadata Database is designed to capture the historical data about the executions of applications. These data are represented in execution contexts, *i.e.*, all information pertinent to a specific execution of an application. Therefore, a number of associations between entities are designed as temporal (characterised by validity – start and end – timestamps).

In the following, we build upon the SENSAPP running example (see Section 3) to exemplify usage of the Metadata Database.

8.1 Running example



Root SLO
SensApp application
Application monitor
Notifier
Servlet Container
Admin
NoSQL DB
Dispatcher
Communication relationship 1
Communication relationship 2
producer
producer
consumer
consumer
SLO on throughput
SLO on response time
Elasticity Rule 1
Elasticity Rule 2
Artefact monitor (response time)
Artefact monitor (throughput)
Elasticity
Elasticity
Elasticity action
Notifier 1
Notifier 2
Tomcat 1
Tomcat 2
Jetty
Admin 1
MongoDB 1
MongoDB 2
Tomcat 3
Tomcat 4
Dispatcher 1
Dispatcher 2
Execution context
Containment relationship 1
Containment relationship 2
Containment relationship 3
Containment relationship 4
Containment relationship 5
Containment relationship 6
Containment relationship 7
Containment relationship 8
Containment relationship 9
Containment relationship 10
Containment relationship 12
Containment relationship 13
Config 1
Config 2
Config 3
Config 4
Config 5
Config 6
Config 7
VM instance 1
VM instance 2
VM instance 3
VM instance 4
VM instance 5
Resource monitor (CPU)
Resource monitor (CPU)
Cloud dependent VM m1.small
Cloud dependent VM 3c4m
Cloud dependent VM large
Execution (time)-dependent information shown in italics
Execution (time)-dependent relationships shown in dashed lines
Cloud provider Amazon
Cloud independent VM small
Cloud provider Flexiant
Cloud independent VM medium
Cloud provider OpenStack
Cloud independent VM large

Figure 16: A sample Metadata Database model for SENSAPP

Figure 16 depicts the Metadata Database model for SENSAPP. This model incorporates elements from the CPSM where instances of the SENSAPP application components along with the corresponding servlet containers and database are deployed on instances of different virtual machines on different cloud providers. The deployment of an applications component on a virtual machine is represented by a temporal associations named "Containment relationship N" rooted at a specific execution context. The communication between application components is represented by a temporal associations named "Communication relationship N". The model also incorporates SLOs on different metrics that are associated to monitoring capabilities (monitoring the corresponding metrics, as well as resource utilisation) and scalability rules that can be invoked in the event of SLO violations.

9 Related work

Deployment and maintenance of distributed systems have been extensively researched over the past decades by various communities including server management systems, distributed systems, and cloud-based applications. To the best of our knowledge, there is not yet any approach combining the strength of recent cloud solutions with the flexibility of MDE.

In the server management community, several solutions, such as IBM Tivoli [12], BCFG2 [13], or CFEngine 3 [9] initially tackled the issue of server (and network) configuration by providing consistent, reproducible and verifiable descriptions of servers configuration. However, by contrast with our approach, these solutions are not tailored to the cloud environment, and do not leverage on infrastructure as a service capabilities.

In the cloud community, several libraries such as jclouds[15], Simple Cloud[16], or DeltaCloud[17] recently emerged to help in reducing cost and effort related to deployment and maintenance of cloud-based applications. While such libraries effectively foster their deployment and maintenance, they remain code-level tools, on which making redesign decisions is difficult and error-prone. Similarly, research projects such as mOSAIC [34], which tackles the vendor lock-in problem by providing an API for provisioning and deployment, are also limited to the code level.

At a higher level of abstraction, more advanced frameworks such as Cloudify[18], Puppet[19] or Chef[20] provide capabilities for the automatic provisioning, deployment, monitoring, and adaptation of cloud systems without being language-dependent. Such solutions provide DSLs to capture and enact cloud-based system management. However, by contrast with our approach, the resulting models are not causally connected to the running system, and may become irrelevant as manual maintenance operations are carried out.

In the models@run-time [8] community, several frameworks already provide causal connections between a running system and its representation as a model. Kevoree [16] provides a first complete models@run-time platform to manage distributed Java applications, but does not leverage infrastructure as a service to operate on cloud-based applications. The work of Shao *et al.* [35] was the first attempt to build a models@run-time platform for the cloud, but remained restricted to monitoring, without providing support for configuration enactment.

Finally, among the standards for cloud computing, the Topology and Orchestration Specification for Cloud Applications (TOSCA) [26] is a related specification developed by the Organization for the Advancement of Structured Information Standards (OASIS). TOSCA provides a language for specifying the components comprising the topology of cloud applications along with the processes comprising their orchestration. Similar to our approach, TOSCA aims at enabling interoperable deployment and management of multi-cloud applications. However, by contrast with our approach, this standard currently lacks a

[15]`http://www.jclouds.org`
[16]`http://simplecloud.org/`
[17]`http://deltacloud.apache.org/`
[18]`http://www.cloudifysource.org/`
[19]`https://puppetlabs.com/`
[20]`http://www.opscode.com/chef/`

models@run-time representation that enables the continuous evolution of multi-cloud applications with no strict boundaries between design-time and run-time activities. The PaaSage consortium plans to contribute to this standard by filling this gap.

10 Large-scale workflow applications, large-scale simulations, and financial auditing applications

The PaaSage Enlarged project introduces three new use cases, namely large-scale scientific workflow applications, large-scale simulations, and financial auditing applications (*cf.* D6.1.3 [22] for a detailed description of these use cases). In this section, we assess how the DSLs adopted in PaaSage are suitable for the specification and execution of these use cases.

10.1 Large-scale workflow applications: HyperFlow

PaaSage Extended envisions the usage of a workflow model and engine for the specification and execution of large-scale scientific workflow applications. For this purpose, we adopt HyperFlow [4], a new workflow execution model and engine inspired by process networks theory and the principles of hypermedia (REST) design.

In HyperFlow, a workflow is defined as a set of *processes* performing well-defined *functions* and exchanging *signals*. In particular, a workflow *process* consists of:

- *Input ports* and associated *signals* which arrive at the process.
- *Output ports* and associated *signals* which are emitted by the process.
- *Function* invoked from the process which transforms input signals to output signals.
- *Type* of the process which determines its general behaviour. For example, a *dataflow* process waits for all data inputs, invokes the function, and emits all data outputs. A *parallel-foreach* process, in turn, waits for any data input, invokes the function, and emits the respective data output.

HyperFlow models can be serialised using a JSON syntax (see Listings 2).

Listing 2: Structure of HyperFlow (JSON syntax)

```
{
  "name": "...",          // name of the workflow
  "functions": [ ... ], // functions used by workflow processes
  "processes": [ ... ], // processes of the workflow
  "signals":   [ ... ], // exchanged signals information
  "schemas":   { ... }, // JSON schemas (for signals) (optional)
  "ins":       [ ... ], // inputs of the workflow (signal names)
  "outs":      [ ... ]  // outputs of the workflow (signal names)
}
```

For instance, let us consider a simple workflow computing a sum of squares of three numbers. Figure 17 illustrates this workflow, while Listing 3 shows its HyperFlow model in JSON.

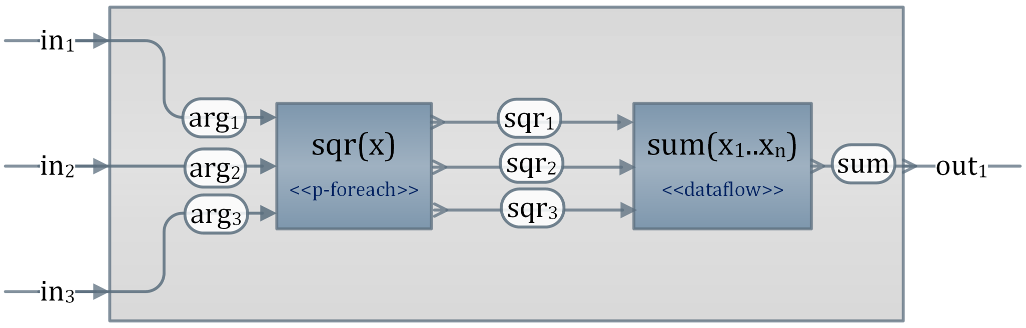


Figure 17: Sum of squares of three numbers in HyperFlow (illustration)

Listing 3: Sum of squares of three numbers in HyperFlow (JSON syntax)

```
{
  "name": "Wf_sqrsum",
  "functions": [ {          // functions used by workflow processes
    "name": "add",
    "module":"functions"
  }, {
    "name": "sqr",
    "module":"functions"
  } ],
  "processes": [ {          // processes of the workflow
    "name": "Sqr",
    "type": "foreach",
    "function": "sqr",
    "ins":  [ "arg1", "arg2", "arg3" ], // signals consumed by this
        process
    "outs": [ "sqr1", "sqr2", "sqr3" ] // signals emitted by this
        process
  }, {
    "name": "Add",
    "type": "dataflow",
    "function": "add",
    "ins": [ "sqr1", "sqr2", "sqr3" ],
    "outs": [ "sum" ]
  } ],
  "signals": [ {
    "name": "arg1",
    "data": [ { "value": 1 } ] // optional signal instances -- will be
         emitted upon workflow start
  }, {
    "name": "arg2",
    "data": [ { "value": 2 } ]
  }, {
    "name": "arg3",
    "data": [ { "value": 3 } ]
  }, {
    "name": "sqr1",
  }, {
    "name": "sqr2",
  }, {
    "name": "sqr3",
  }, {
    "name": "sum",
  } ],
  "ins": [ "arg1", "arg2", "arg3" ],
  "outs": [ "sum" ]
}
```

Another example shows a fragment of the HyperFlow model for a Montage [7] scientific workflow which produces very large mosaic images of the sky (Listing 4). This particular workflow has over 10,000 processes and more than 23,000 signals which represent input, output and intermediate files. All workflow processes invoke the same function: `amqp_command`, which passes over the execution to a cloud. To this end, the function generates a task specification, sends it to a message queue, and awaits on the same queue for the completion of the task. Local executors deployed in the cloud take care of fetching the tasks, executing them, and notifying about their completion. In this simple but effective way the workflow model and coordination in HyperFlow is decoupled from the underlying task execution model.

Listing 4: Montage scientific workflow expressed in the HyperFlow language (fragment)

```
{
  "name": "Montage_10k",
  "functions": [ {
    "name": "amqp_command", // sends a task to the cloud via a message
         queue
    "module": "functions"
  } ],
  "processes": [ {
    "name": "mProjectPP",
    "function": "amqp_command",
    "config": {
       "executable": "mProjectPP",
       "args": " ... "
    },
    "ins": [ 0, 3 ],
    "outs": [ 1, 2 ]
  },
  // ... about 10k more processes ...
  {
    "name": "mJPEG", // last task: generation of the final image
    "function": "command",
    "executor": "syscommand",
    "config": {
       "executable": "mJPEG",
       "args": "-ct 1 -gray shrunken_20090720_143653_22436.fits -1.5s
           60s gaussian -out shrunken_20090720_143653_22436.jpg"
    },
    "ins": [ 22960 ],
    "outs": [ 22961 ]
  } ],
  "signals": [ { // here signals represent files and contain only file
       names
     "name": "2mass-atlas-980529s-j0150174.fits"
  },
  // ... about 23k more signals
  } ],
  "ins": [ ... ],
  "outs": [ ... ]
}
```

PaaSage adopts a set of DSLs to describe the provisioning and deployment of HyperFlow as well as the requirements for large-scale scientific workflow applications. In the following, we present an example of how these DSLs can be used for this purpose.

Provisioning and deployment

The provisioning and deployment of large-scale scientific workflow applications can be specified using CLOUDML (see Section 4). In workflow applications, individual steps or a cluster of steps may be architected in such a way that they can be processed independently and on separate nodes. In these cases, the workflow can be mapped to a set of deployable artefacts. These artefacts can be regarded as application components, although the relationships between these components are constrained according to the actual workflow model. CLOUDML can then be used to specify the specific provisioning and deployment for the processing of these. For the next phase of the work in WP2 we will investigate how to further support the architecting and mapping of large-scale scientific workflow models in this way.

Figure 6 shows an example of CPIM of a scientific workflow application using the CLOUDML visual syntax (see Figure 5). This CPIM consists of: a master Workflow Engine node with the HyperFlow workflow engine, a database (Redis), a messaging server (RabbitMQ); a storage node with a storage server (NFS server); and worker node with the Executor and application binaries.

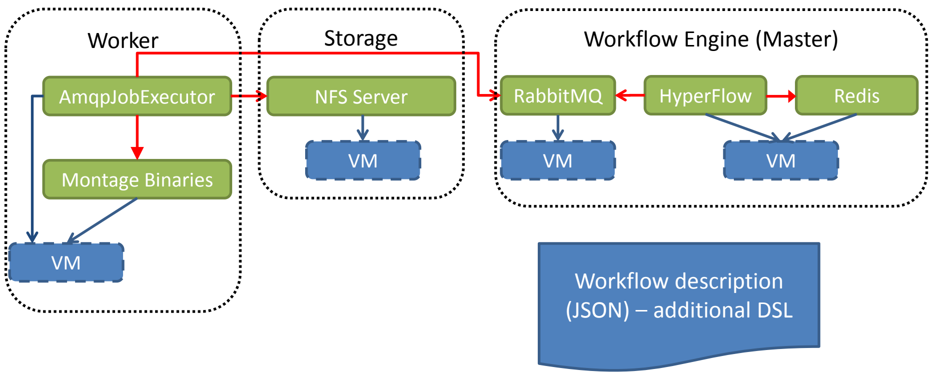


Figure 18: A sample CPIM for workflow application in hyperflow

Figure 19 shows the CPSM of the workflow application, where the master Workflow Engine node is deployed to a private AGH (OpenStack) cloud in Poland while the Storage and Worker nodes are deployed on Amazon EC2 in the EU.

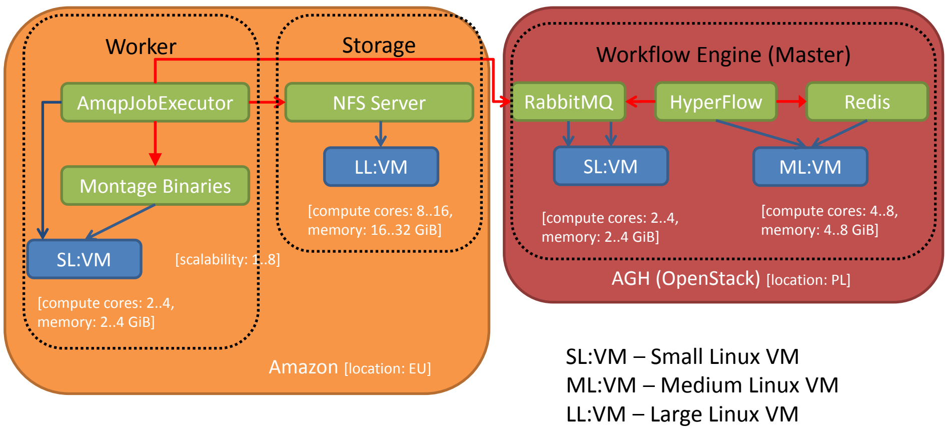


Figure 19: A sample CPSM for workflow application in hyperflow

Figure 20 shows the CPSM of the workflow application after a horizontal scaling took place, where an additional Worker node is deployed on Amazon EC2 cloud.

Requirements and SLAs

The requirements on infrastructure resource for large-scale scientific workflow applications can be specified using Saloon (see Section 5). Table 2 shows some requirements on operating system, number of compute cores, amount of memory,

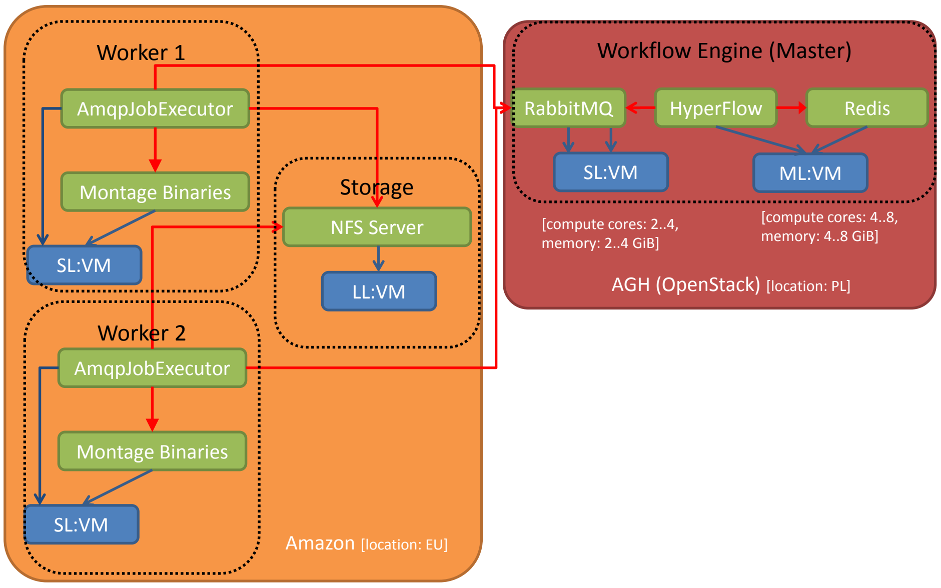


Figure 20: A sample CPSM for workflow application in hyperflow after autoscaling

and deployment model. These requirements are then matched with actual VM types offered by cloud providers (*e.g.* `m1.small` on Amazon EC2).

In addition to the requirements on the infrastructure resources, large-scale scientific workflow applications may have SLOs and QoS constraints. The following are three sample scenarios:

- Scenario 1:
  Deadline: 6 hours
  Minimize: cost

- Scenario 2:
  Minimize: time

- Scenario 3:
  Maximize: resource utilization

On the one hand, the objectives and constraints in Scenarios 1 and 2 are intended mainly to be used at the planning phase of the workflow execution. They are used by the Worklfow Planner module of HyperFlow [22] to prepare a scheduling plan in order to satisfy these constraints. On the other hand, the objective in Scenario 3 is application-independent and can be used for dynamic provisioning when the size of the cluster of workers is scaled automatically to

Table 2: Examples of requirements for scientific workflows

AmqpJobExecutor • OS: Ubuntu Linux • Compute cores: 1 • Memory: 2 GiB • Deployment model: public	Montage Binaries • OS: Ubuntu Linux • Compute cores: 1 • Memory: 2 GiB • Deployment model: public
NFS Server • OS: Red Hat Enterprise Linux • Compute cores: 2 • Memory: 4 GiB • Deployment model: private	RabbitMQ • OS: Ubuntu Linux • Compute cores: 1 • Memory: 2 GiB • Deployment model: private
HyperFlow Engine • OS: Ubuntu Linux • Compute cores: 2 • Memory: 4 GiB • Deployment model: private	Redis Database • OS: Ubuntu Linux • Compute cores: 2 • Memory: 4 GiB • Deployment model: private

maximize the resource utilization. These objectives and constraints may be specified using WS-Agreement, although further investigation will be performed after *month 13* in order to assess how these scenarios can be translated to concrete WS-Agreement syntax.

10.2 Large-scale simulations: Scalarm

PaaSage Enlarged envisions the usage of a data farming solution for large-scale simulations. For this purpose, we adopt Scalarm[21]. Scalarm is a distributed platform consisting of several services, each with a clearly defined purpose:

[21] `http://www.scalarm.com/`

- *Experiment Manager* is responsible for managing the workflow of data farming experiments. It exposes a graphical user interface to prepare parameter space for an experiment, schedule and monitor computation on different infrastructures, and analyse collected output from executed simulations.
- *Storage Manager* provides a programming interface to store and query simulation results, in both structural and binary forms. In addition it is responsible for exposing a MongoDB cluster.
- *Simulation Manager* is a wrapper of a simulation "job". It is responsible for reserving computational resources in the cloud to run the actual simulations (or any other programs). It communicates with the Experiment Manager to get input parameters for subsequent simulation runs.
- *Information Service* is a "well-known" place where all above Scalarm services are registered. In addition, any service can query the Information Service about location of other Scalarm services.

PaaSage adopts a set of DSLs to describe the provisioning and deployment of Scalarm as well as the requirements for large-scale simulations. In the following, we present an example of how these DSLs can be used for this purpose.

Provisioning and deployment

The provisioning and deployment of Scalarm and other artefacts required to perform a large-scale simulation can be specified using CLOUDML (see Section 4). Figure 21 shows a CPIM using the CLOUDML visual syntax (see Figure 5). This CPIM consists of Scalarm as well as other artefacts required to perform a numerical simulation of a metallurgical process. This experiment aims at exploring a large parameter space of possible simulation inputs. Each element of such a parameter space describes a single variant of a simulation.

An important feature of PaaSage is the support for cloud bursting, *i.e.* "bursting" the additional workload to an external cloud on an on-demand basis, when the computing resources in the internal cloud are running out. Figure 20 shows a possible CPSM after a cloud bursting, where the artefacts executing the simulation have been moved to a Amazon EC2 cloud in the EU.

The cloud bursting can exploit multiple clouds too. Figure 20 shows a possible CPSM after a second cloud bursting, where the artefacts executing the simulation have been replicated in a Flexiant cloud in the UK.

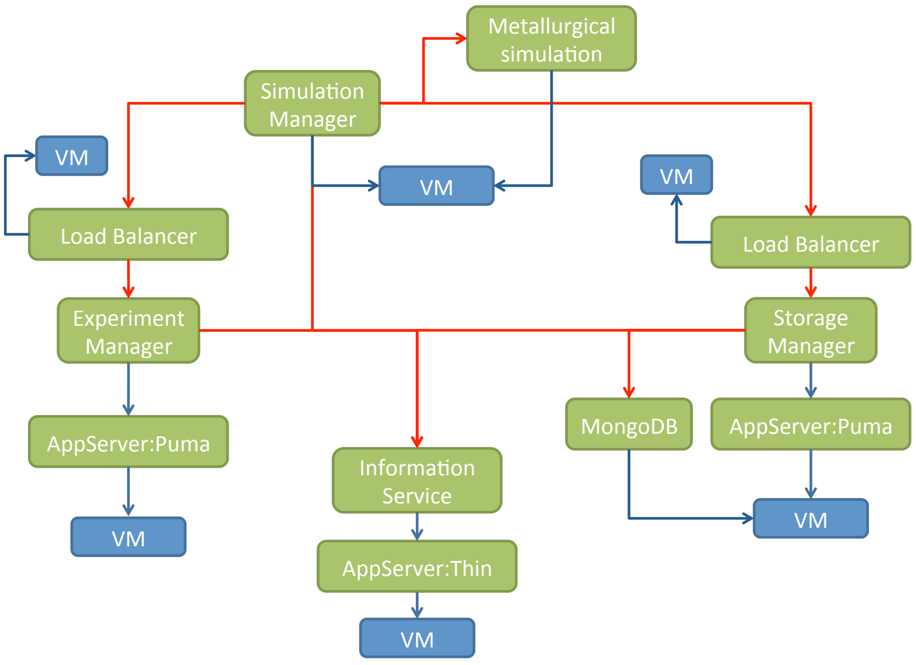


Figure 21: A deployment diagram of Scalarm as Cloud Provider Independent Model.

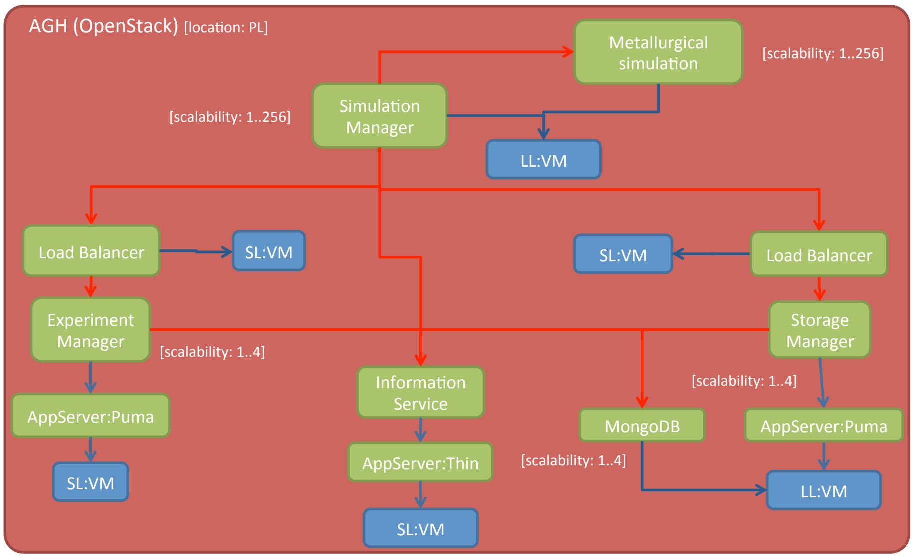


Figure 22: A possible actual deployment of Scalarm in a multi-cloud environment as Cloud Provider Specific Model.

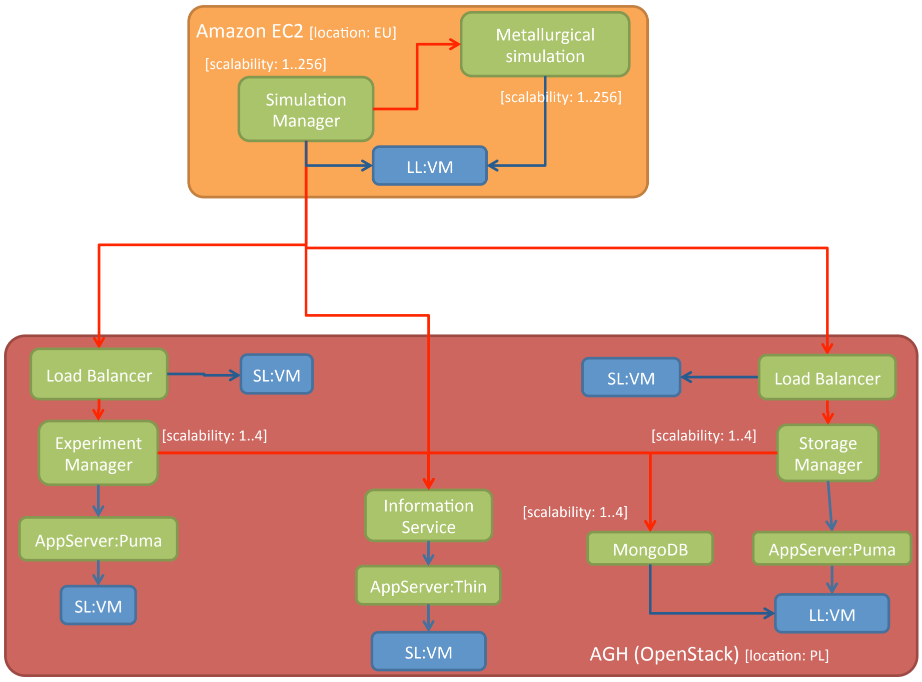


Figure 23: A snapshot of possible Scalarm deployment after autoscaling.

Requirements

The requirements on infrastructure resources large-scale simulations can be specified using Saloon (see Section 5). Table 3 shows some requirements on operating system, number of compute cores, amount of memory, and deployment model.

10.3 Financial auditing applications

The objective of the new financial case study provided by UCY and IBSAC (*cf.* D6.1.3 [22]) is to port the Audit application to the cloud utilizing the MDE methodology and environment defined and developed in PaaSage project. The requirements of the financial sector application are similar and thus addressed as part of other applications described in the deliverable. These requirement including portability across multiple clouds to avoid data/vendor lock-in, migration between multiple providers, data security, as well as load-balancing and autoscaling mechanisms. Based on the analysis of the other case studies we conclude that the proposed MDE approach and the languages described in this document are suitable for the specification and execution of the financial case study.

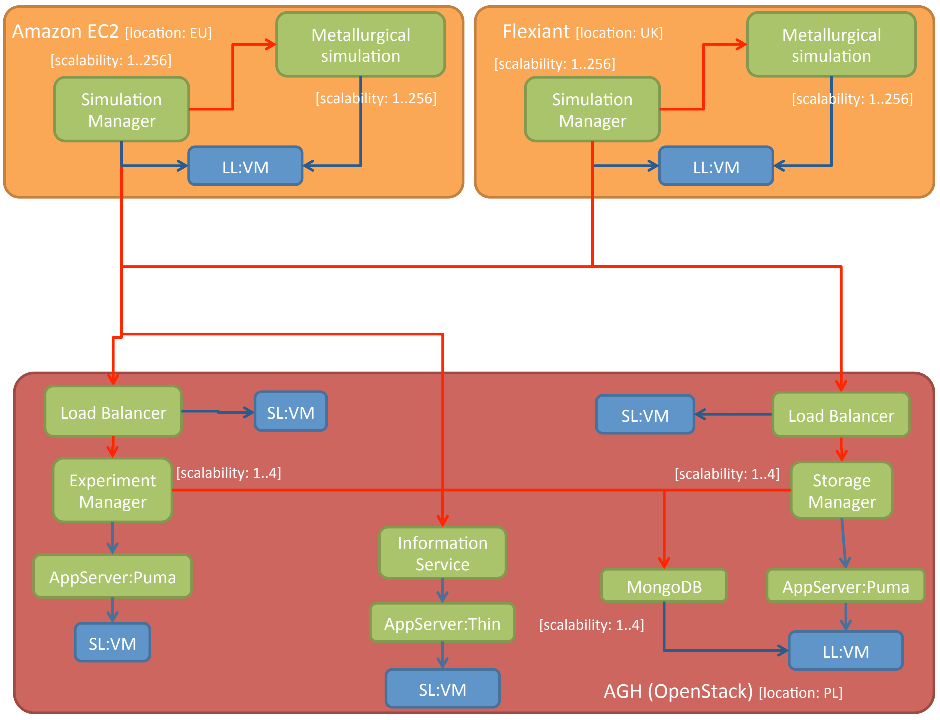


Figure 24: A snapshot of possible Scalarm deployment after a manual increase of computational resources.

10.4 Summary

In this section, we gave an overview of how the large-scale scientific workflow applications in HyperFlow and large-scale simulations in Scalarm can leverage upon the set of DSLs adopted in PaaSage. HyperFlow targets workflow concerns that are application-specific, whereas the other DSLs target provisioning and deployment, requirements and goals, and SLAs. At this stage of the project, we can assess that the proposed DSLs cover all the relevant aspects of the specification and execution of cloud-based applications, including the one from the new use cases. However, this will be further assessed after *month 13* during the development of the prototype.

Table 3: Examples of requirements for Scalarm

Experiment Manager • OS: Ubuntu Linux • Compute cores: 2 • Memory: 4 GiB • Deployment model: private	Storage Manager • OS: Ubuntu Linux • Compute cores: 4 • Memory: 8 GiB • Deployment model: private
MongoDB • OS: Ubuntu Linux • Compute cores: 8 • Memory: 8 GiB • Deployment model: private	Simulation Manager • OS: Ubuntu Linux • Compute cores: 1 • Memory: 1 GiB • Deployment model: public
Metallurgical simulation • OS: Ubuntu Linux • Compute cores: 1 • Memory: 1 GiB • Deployment model: public	Information Service • OS: Ubuntu Linux • Compute cores: 1 • Memory 2 GiB • Deployment model: private

11 Conclusions and Future Work

In this deliverable, we have discussed how the DSLs adopted in PaaSage can enable dynamically adaptive multi-cloud applications by leveraging upon MDE techniques and methods. The DSLs facilitate the specification of different cloud concerns of multi-cloud applications at all life-cycle phases of configuration, deployment, and execution. The Metadata Database provides a unified representation of these cloud concerns that facilitates reasoning, simulation, and enactment of adaptation actions.

Please note the capabilities of the DSLs presented in this deliverable reflect our understanding of the requirements of PaaSage at month 12. These requirements will be developed iteratively throughout the course of the project. Therefore, an important task is to adapt the capabilities of the DSLs to the changing requirements, and adapt the Metadata Database schema accordingly. In this respect, the research partners in PaaSage will provide feedback on whether the elements of each DSL are adequate to develop the components of the PaaSage platform. Similarly, the industrial partners in PaaSage will provide feedback on whether the elements of each DSL are satisfactory for modelling the use cases.

References

[1] Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke and Ming Xu. *Web Services Agreement Specification (WS-Agreement)*. Tech. rep. Open Grid Forum, Mar. 2007.

[2] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, 2009. ISBN: 9780521125499.

[3] Colin Atkinson and Thomas Kühne. "Rearchitecting the UML infrastructure". In: *ACM Transactions on Modeling and Computer Simulation* 12.4 (2002), pp. 290–321. DOI: 10.1145/643120.643123.

[4] Bartosz Baliś. "Hypermedia Workflow: A New Approach to Data-Driven Scientific Workflows". In: *SC Companion*. IEEE Computer Society, 2012, pp. 100–107. ISBN: 978-1-4673-6218-4. DOI: 10.1109/SC.Companion.2012.25.

[5] David Benavides, Pablo Trinidad Martín-Arroyo and Antonio Ruiz Cortés. "Automated Reasoning on Feature Models". In: *CAiSE 2005: 17th International Conference Advanced Information Systems Engineering*. Ed. by Oscar Pastor and João Falcão e Cunha. Vol. 3520. Lecture Notes in Computer Science. Springer, 2005, pp. 491–503. ISBN: 3-540-26095-1. DOI: 10.1007/11431855_34.

[6] David Benavides, Sergio Segura and Antonio Ruiz Cortés. "Automated analysis of feature models 20 years later: A literature review". In: *Inf. Syst.* 35.6 (2010), pp. 615–636. DOI: 10.1016/j.is.2010.01.001.

[7] G. B. Berriman, Ewa Deelman, John C. Good, Joseph C. Jacob, Daniel S. Katz, Carl Kesselman, Anastasia C. Laity, Thomas A. Prince, Gurmeet Singh and Mei-Hu Su. "Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand". In: *Proceedings of SPIE* 5493 (2004), pp. 221–232. DOI: 10.1117/12.550551.

[8] Gordon S. Blair, Nelly Bencomo and Robert B. France. "Models@run.time". In: *IEEE Computer* 42.10 (2009), pp. 22–27. DOI: 10.1109/MC.2009.326.

[9] Mark Burgess and Ricky Ralston. "Distributed Resource Administration Using Cfengine". In: *Softw., Pract. Exper.* 27.9 (1997), pp. 1083–1101. DOI: 10.1002/(SICI)1097-024X(199709)27:9%3C1083::AID-SPE126%3E3.0.CO;2-H.

[10] Oscar Corcho, Mariano Fernández-López and Asunción Gómez-Pérez. "Ontological Engineering: Principles, Methods, Tools and Languages". In: *Ontologies for Software Engineering and Software Technology*. Ed. by Coral Calero, Francisco Ruiz and Mario Piattini. Springer Berlin Heidelberg, 2006, pp. 1–48. ISBN: 978-3-540-34517-6. DOI: 10.1007/3-540-34518-3_1.

[11] Krzysztof Czarnecki, Simon Helsen and Ulrich W. Eisenecker. "Formalizing Cardinality-based Feature Models and their Specialization". In: *Software Process: Improvement and Practice* 10.1 (2005), pp. 7–29. DOI: 10.1002/spip.213.

[12] Thomas Delaet, Wouter Joosen and Bart Vanbrabant. "A survey of system configuration tools". In: *LISA 2010: 24th international conference on Large installation system administration*. USENIX Association, 2010, pp. 1–8.

[13] Narayan Desai et al. "A Case Study in Configuration Management Tool Deployment". In: *LISA 2005: 19th Conference on Systems Administration*. USENIX, 2005, pp. 39–46.

[14] Nicolas Ferry, Franck Chauvel, Alessandro Rossini, Brice Morin and Arnor Solberg. "Managing multi-cloud systems with CloudMF". In: *NordiCloud 2013: 2nd Nordic Symposium on Cloud Computing and Internet Technologies*. Ed. by Arnor Solberg, Muhammad Ali Babar, Marlon Dumas and Carlos E. Cuesta. ACM, 2013, pp. 38–45. ISBN: 978-1-4503-2307-9. DOI: 10.1145/2513534.2513542.

[15] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin and Arnor Solberg. "Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems". In: *CLOUD 2013: IEEE 6th International Conference on Cloud Computing*. Ed. by Lisa O'Conner. IEEE Computer Society, 2013, pp. 887–894. ISBN: 978-0-7685-5028-2. DOI: 10.1109/CLOUD.2013.133.

[16] François Fouquet, Erwan Daubert, Noël Plouzeau, Olivier Barais, Johann Bourcier and Jean-Marc Jézéquel. "Dissemination of Reconfiguration Policies on Mesh Networks". In: *DAIS 2012: 12th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*. Ed. by Karl M. Göschka and Seif Haridi. Vol. 7272. Lecture Notes in Computer Science. Springer, 2012, pp. 16–30. ISBN: 978-3-642-30822-2. DOI: 10.1007/978-3-642-30823-9_2.

[17] Thomas R. Gruber. "A translation approach to portable ontology specifications". In: *Knowledge Acquisition* 5.2 (June 1993), pp. 199–220. ISSN: 1042-8143. DOI: 10.1006/knac.1993.1008.

[18] Keith Jeffery and Tom Kirkham. *D1.6.1 – Initial Architecture Design*. PaaSage project deliverable. Oct. 2013.

[19] Steffen Kächele, Christian Spann, Franz J. Hauck and Jörg Domaschka. “Beyond IaaS and PaaS: An Extended Cloud Taxonomy for Computation, Storage and Networking”. In: *UCC 2013: IEEE/ACM 6th International Conference on Utility and Cloud Computing*. To appear. IEEE Computer Society, 2013.

[20] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak and A. Spencer Peterson. *Feature-Oriented Domain Analysis (FODA) - Feasibility Study*. Tech. rep. The Software Engineering Institute, 1990. URL: `http://www.sei.cmu.edu/reports/90tr021.pdf`.

[21] Jeffrey O. Kephar and David M. Chess. “The vision of autonomic computing”. In: *Computer* 36.1 (Jan. 2003), pp. 41–50. DOI: `10.1109/MC.2003.1160055`.

[22] Maciej Malawski, Bartosz Baliś, Dariusz Król, Achilleas Achilleos, Christos Mettouris and Avgoustinos Costantinides. *D6.1.3 – Initial Requirements (Extended)*. PaaSage project deliverable. Nov. 2013.

[23] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. Special Publication 800-145. National Institute of Standards and Technology, Sept. 2001.

[24] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey and Arnor Solberg. “Models@Run.time to Support Dynamic Adaptation”. In: *IEEE Computer* 42.10 (2009), pp. 44–51. DOI: `10.1109/MC.2009.327`.

[25] Sébastien Mosser, Franck Fleurey, Brice Morin, Franck Chauvel, Arnor Solberg and Iokanaan Goutier. “SENSAPP as a Reference Platform to Support Cloud Experiments: From the Internet of Things to the Internet of Services”. In: *SYNASC 2012: 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE Computer Society, 2012, pp. 400–406. ISBN: 978-1-4673-5026-6. DOI: `10.1109/SYNASC.2012.71`.

[26] Derek Palma and Thomas Spatzier. *Topology and Orchestration Specification for Cloud Applications (TOSCA)*. Tech. rep. Organization for the Advancement of Structured Information Standards (OASIS), June 2013. URL: `http://docs.oasis-open.org/tosca/TOSCA/v1.0/cos01/TOSCA-v1.0-cos01.pdf`.

[27] Antonis Papaioannou and Kostas Magoutis. “An Architecture for Evaluating Distributed Application Deployments in Multi-Clouds”. In: *CloudCom 2013: IEEE 5th International Conference on Cloud Computing Technology and Science*. 2013.

[28] Antoine Pichot, Philipp Wieder, Oliver Wäldrich and Wolfgang Ziegler. "Dynamic SLA-negotiation based on WS-Agreement". In: *Technical Report TR-0082* (June 2007).

[29] Clément Quinton, Nicolas Haderer, Romain Rouvoy and Laurence Duchien. "Towards multi-cloud configurations using feature models and ontologies". In: *MultiCloud 2013: International Workshop on Multi-cloud Applications and Federated Clouds*. ACM, 2013, pp. 21–26. ISBN: 978-1-4503-2050-4. DOI: `10.1145/2462326.2462332`.

[30] Clément Quinton, Daniel Romero and Laurence Duchien. "Cardinality-based feature models with constraints: a pragmatic approach". In: *SPLC 2013: 17th International Software Product Line Conference*. Ed. by Tomoji Kishi, Stan Jarzabek and Stefania Gnesi. ACM, 2013, pp. 162–166. ISBN: 978-1-4503-1968-3. DOI: `10.1145/2491627.2491638`.

[31] Clément Quinton, Romain Rouvoy and Laurence Duchien. "Leveraging Feature Models to Configure Virtual Appliances". In: *CloudCP 2012: 2nd International Workshop on Cloud Computing Platforms*. ACM, 2012, 2:1–2:6. ISBN: 978-1-4503-1161-8. DOI: `10.1145/2168697.2168699`.

[32] Alessandro Rossini and the PaaSage consortium. *D2.1.2 – CloudML Implementation Documentation - First version*. PaaSage project deliverable. Apr. 2014.

[33] Alessandro Rossini, Arnor Solberg, Daniel Romero, Jörg Domaschka, Kostas Magoutis, Lutz Schubert, Nicolas Ferry and Tom Kirkham. *D2.1.1 – CloudML Guide and Assesment Report*. PaaSage project deliverable. Oct. 2013.

[34] Calin Sandru, Dana Pectu and Victor Ion Munteanu. "Building an Open-Source Platform-as-a-Service with Intelligent Management of Multiple Cloud Resources". In: *UCC 2012: IEEE 5th International Conference on Utility and Cloud Computing*. IEEE Computer Society, 2012, pp. 333–338. ISBN: 978-1-4673-4432-6. DOI: `10.1109/UCC.2012.54`.

[35] Jin Shao, Hao Wei, Qianxiang Wang and Hong Mei. "A Runtime Model Based Monitoring Approach for Cloud". In: *CLOUD 2010: IEEE 3rd International Conference on Cloud Computing*. IEEE Computer Society, 2010, pp. 313–320. ISBN: 978-1-4244-8207-8. DOI: `10.1109/CLOUD.2010.31`.