Cloud Orchestration at the Level of Application

Project Acronym: **COLA**

Project Number: **731574**

Programme: **Information and Communication Technologies**
Advanced Computing and Cloud Computing

Topic: **ICT-06-2016 Cloud Computing**

Call Identifier: **H2020-ICT-2016-1**
Funding Scheme: **Innovation Action**

Start date of project: 01/01/2017 Duration: 30 months

Deliverable:

D5.1 Analysis of existing application description approaches

Due date of deliverable: 31/03/2017 Actual submission date: 31/03/2017

WPL: Gabriele Pierantoni

Dissemination Level: PU

Version: WIP

1. Table of Contents

2. List of Figures and Tables

Figures

Tables

3. Status, Change History and Glossary

Status:	Name:	Date:	Signature:
Draft:	Gabriele Pierantoni	06/02/17	Gabriele Pierantoni
Reviewed:	Tamas Kiss	23/03/17	Tamas Kiss
Approved:	Tamas Kiss	31/03/17	Tamas Kiss

Table 1, Status Change History

Version	Date	Pages	Author	Modification
V0.1	06/02	ALL	G. Pierantoni	Empty Skeleton
V0.2	20/02	ALL	G. Pierantoni, G. Terstyanszky	Added compared analysis of TOSCA, Azure, Amazon and ORACLE
V1.0	07/03	ALL	G. Pierantoni	Added sections about CAMP and TOSCA security
V1.1	20/03	ALL	G. Pierantoni	Draft for Review
v2.0	27/03	All	G. Terstyanszky	Reviewing the report
V2.1	29/03	All	G. Terstyanszky	Added missing section, Introduction, Orchestrator Languages

Table 2, Deliverable Change History

Glossary

API	Application Programming Interface
CAMP	Cloud Application Management for Platforms
COLA	Cloud Orchestration at the level of Application
IAAS	Infrastructure as a Service
PAAS	Platform as a Service
SAAS	Software as a Service
TOSCA	Topology Orchestration Specification for Cloud Application

Table 3, Glossary

4. Introduction

DoW specifies D5.1 Analysis of existing application description approaches deliverable as follows:

> "This deliverable will give a summary of the investigation on the existing application description approaches. It will also shortly describe the selected application description approach and explain why it was selected."

This deliverable aims at analysing the existing application descriptions. This state of the art overview is fundamental to decide which description language and overall design approach is best suited to solve COLA's requirements. Deliverable 5.1 constitutes a first step in the definition and implementation of the COLA Application Description Templates, therefore, it is fundamental to assess all viable existing solutions and match them against COLA's specific requirements.

In order to achieve this, D5.1, is structured as follows:

- **Chapter 5** – gives a short overview of the concept that COLA will apply to describe applications, outlining how it specifies architecture, service and implementation levels.
- **Chapter 6** – briefly describes major platform-oriented and platform independent application description languages that are perspective candidates to be used in COLA.
- **Chapter 7** – investigates application description approaches used by major cloud orchestration tools.
- **Chapter 8** – defines the comparison criteria and recapitulates strengths and weaknesses of each of the perused approaches with regard of the COLA requirements
- **Chapter 9** –describes the final decision on the application description solution to be used in COLA and lists reasons that justify this decision.

5. COLA Application Description Concept

To support efficient orchestration of application execution on the Cloud, COLA will elaborate the concept of application template to support application description at three levels: architecture, service, implementation levels. The architecture level manages architectures that can be used by different applications in business, industry and public sector. Application templates describe these architectures specifying their service types, relations, and requirements. The service types are high-level services, for example business logic, presentation logic, data service, etc. The service level identifies particular types of services specified in the application template. For example, either MongoDB, SQL or other database can be used as data service. Service descriptions must be added to the application template to create a service template. The implementation level specifies the service version needed to run the service, for example MongoDB v3.1, v3.2 etc., and the required service signature. Adding this information to the service template they create an implementation template. Each application template may have *"1…k"* service templates, and each of them may have *"1…m"* implementation templates (see Figure 1).

To deploy and run an application in the Cloud, application developers working in business, industry and public sector, and description developers, who are experts in application descriptions and in the Cloud, must cooperate. They follow top-down approach. First, they elaborate the application templates. Next, they define the service and implementation templates describing services and their implementations. Finally, they publish these templates in a repository.

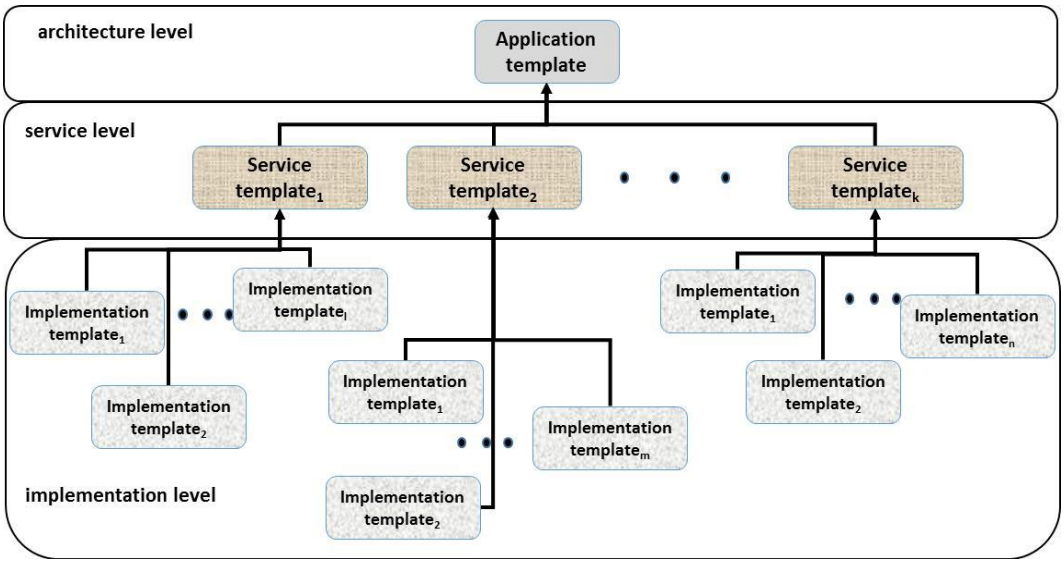

Figure 1, Specifying applications to be deployed and run on the cloud

Publishing these templates will significantly improve application development and sharing because application developers can select and use the existing templates. They can search the repository for application templates they need. Finding required templates they check whether there are service and implementation templates they need. For example, if developers were able to find an existing application and a service template needed but no proper implementation template is available, they can re-use the first two templates and has to create only the third one.

6. Overview of Existing Application Description Approaches

WP5 analysed existing platform specific and platform independent application description approaches to select a service and implementation level description that can be extended with the architecture level description. After making this decision, WP5 will define the concept of the application template to describe the architecture level, develop its formal description, and define how it can be integrated with service and implementation templates. In the formal description there will be a special emphasis on how to specify QoS parameters (maximum response time, minimum usage time, throughput, security policies and credentials, etc.) to support scaling up and down the services as required, and to manage security in different application scenarios. COLA will also extend the formal description of all three templates with metadata to support efficient search and selection of these templates.

6.1 Platform specific application description approaches

6.1.1 Amazon Solutions

Amazon uses Amazon Machine Image (AMI) template [1] to describe all information required to launch an Amazon EC2 instance. An AMI template includes

- **root volume** for the instance, i.e. an operating system, an application server, and applications,
- **launch permissions** that control which AWS accounts can use the AMI to launch instances and,
- **block device mapping** that specifies the volumes to attach to the instance when it's launched.

The AMI template must contain at least the base operating system. It may also include additional configuration and software code. Launching an instance from a base AMI containing only the operating system the base AMI can be further customized adding additional configuration data and software after it has been launched. Customized AMI contains the complete software stack.

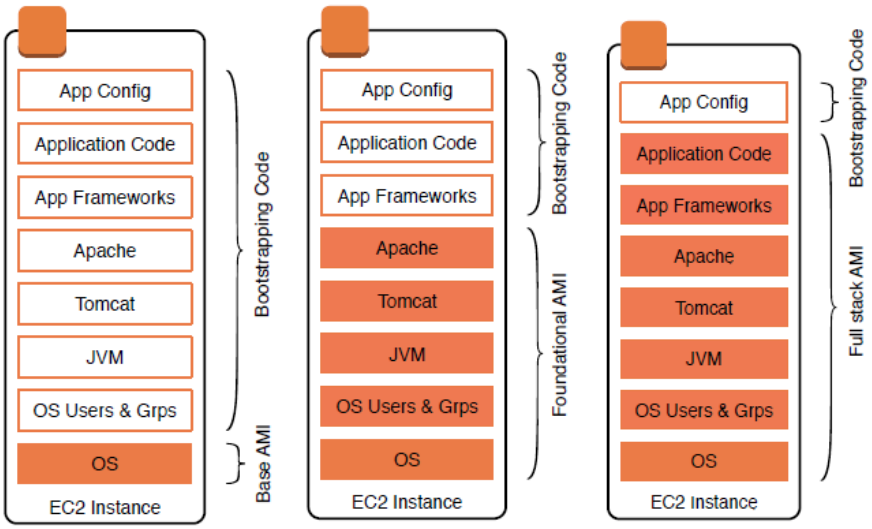

Figure 2, Base, foundational and customized AMI template

There are three AMI templates (see Figure 2):

- **Base AMI template** contains only the OS image.
- **Foundational AMI template** includes elements of a stack that change infrequently for example, JVM, application server etc.
- **Full stack AMI template** contains all elements of the stack. It is recommended to use if the application changes infrequently, or if it has rapid auto scaling requirements.

Amazon CloudFormation[2] supports development, deployment and running applications on the Amazon cloud. The applications are described by the AWS CloudFormation templates that combine AMI templates. The templates are stored as text files that comply with the JavaScript Object Notation (JSON) or YAML. The templates can be created and edited in any text editor and can be managed in the source code IDE. The templates specify the AWS resources that make up the application stack and that must be created and configured. They specify an object as a name-value pair or a pairing of a name with a set of child objects enclosed. The only required top-level object is the resources object, which must declare at least one resource. Amazon provides the CloudFormation Tool through the AWS CloudFormation Designer to develop the CloudFormation templates. It creates an AWS CloudFormation template from existing AWS resources in the developer's account. The developer has to select any supported AWS resources that are running in her/his account, and the CloudFormation Tool creates a template in an Amazon S3 bucket. AWS CloudFormation uses these templates as blueprints for building AWS resources.

6.1.2 Microsoft Azure

Microsoft Azure[3] describes resources through Azure Resource Manager (ARM). ARM combines together compute, storage and network resources and shows them as a single unit that can be created, managed and deleted together. ARM enables template based Azure resource deployment. ARM templates contain four entities (see Figure 3):

1. **parameters** that can be entered during run-time with set of values or default values pre-defined,
2. **variables** that are static in the code and used for deploying resources,
3. **resources** to be deployed, and,
4. **outputs** to be produced.

There are four template scopes.

1. **capacity scope** delivers a set of resources in a standard topology that is pre-configured to be in compliance with regulations and policies.
2. **capability scope** is focused on deploying and configuring a topology for a given technology for example SQL Server, Cassandra, Hadoop, etc…
3. **end-to-end solution scope** is targeted beyond a single capability, and instead focused on delivering an end to end solution comprised of multiple capabilities.
4. **solution scope** manifests itself as a set of one or more capability scoped templates with solution specific resources, logic, and desired state, for example Kafka, Storm, and Hadoop.

ARM templates enable tagging resources with key/value pairs to further categorize and view them across resource groups and within the portal, and across subscriptions. Tagging adds metadata about a resource. ARM templates allow managing resource groups that include all resources needed for an application. ARM templates support deploying topologies by managing resources required for applications using resource groups that are logical

containers for grouping Azure resources. They apply role-based access control (RBAC) to grant access to users, groups and services. ARM templates can be decomposed to provide a modular approach to template development to supports reuse, extensibility, testing, and tooling. There are several sub-templates such as main sub-template, member sub-template, optional resources and shared resources sub-template, etc. Azure Resource Manager uses Azure Key Vault to orchestrate and store VM secrets and certificates. The ARM templates only contain URI references to the secrets that are stored in the Key Vault and are under full RBAC control of a trusted operator.

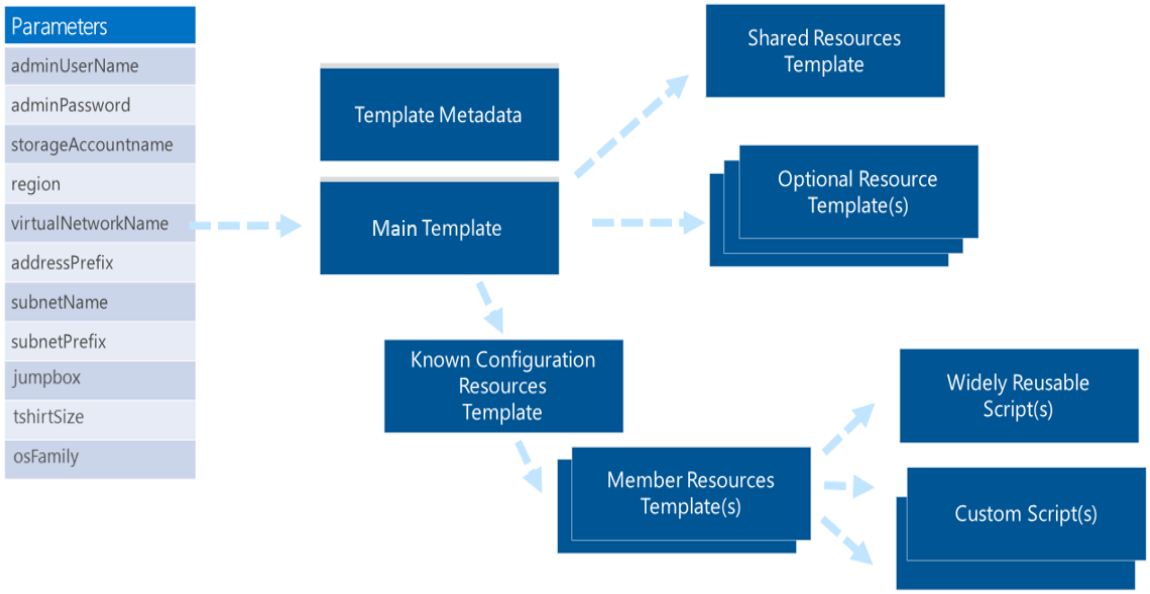

Figure 3, Azure Resource Manager Template

6.1.3 ORACLE

ORACLE enables quick configuration and provisioning of multi-tier application topologies onto virtualized and cloud environments by capturing the configuration and packaging of existing software components as self-contained building blocks known as appliances. These appliances can then be easily connected to form application blueprints, called as assemblies [4] and [5]. They are built on Oracle VM Templates that allow deploying a fully configured software stack by offering pre-installed and pre-configured software images. The Template contains the virtual machine configuration information, virtual disks that contain the operating system and any application software. These components are packaged together as an Oracle VM Template file according to the industry-standard Open Virtualization Format (OVF) standard. While Oracle VM Template manages one virtual machine usually for a single tier application, Oracle Virtual Assembly is a collection of interrelated software appliances that are automatically configured to work together. Assembly is a set of appliance that includes one or more templates plus the meta-data that describes the relations and connection among the appliances. When the assembly is deployed its templates are deployed as virtual machines. The assemblies can be quickly instantiated into a collection of interrelated virtual machines running on a virtualized pool of servers, with all virtual machine instances configured and wired to communicate with each other automatically. ORACLE offers developers the Oracle Virtual Assembly Builder (OVAB) [6] to support customisation and provisioning of complex enterprise applications with no manual intervention onto virtualized and cloud environments.

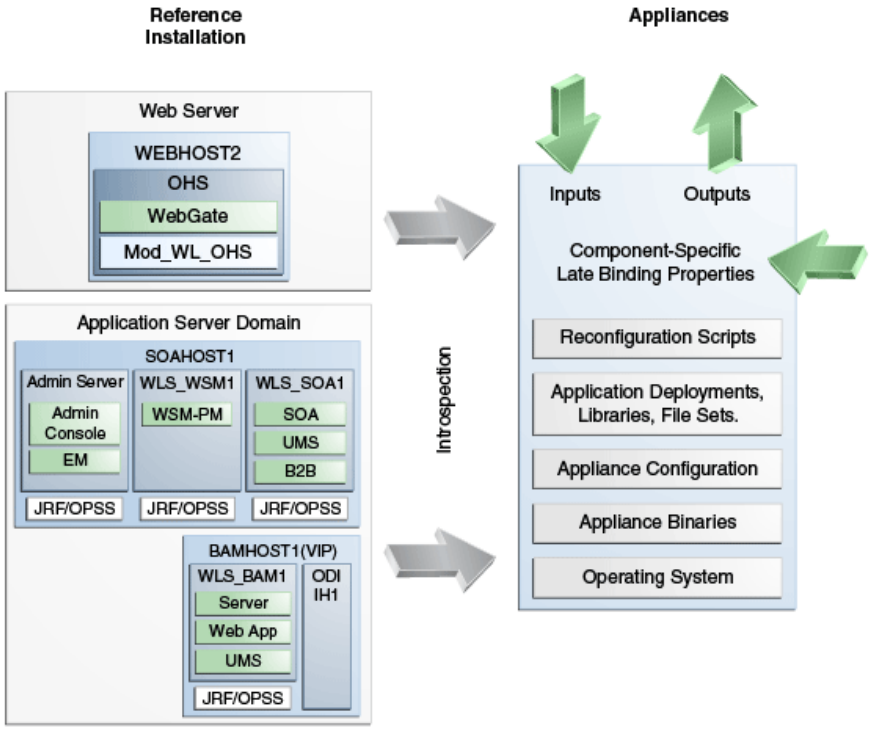

Figure 4, OVAB Assembly Template

6.2 Platform independent application description approaches

6.2.1 TOSCA

An OASIS technical committee [7], containing industrial partners, service providers and research organizations has developed the TOSCA (Topology and Orchestration Specification for Cloud Applications) Language Specification [8], [9], [10] and [11] as an interface interoperability standard[12]. Its main goal is to enable the creation of portable cloud applications and the automation of their deployment and management. In order to achieve this goal, TOSCA focuses on the following three goals [8]:

1. **Automated application deployment and management**. TOSCA aims at providing a language to express how to automatically deploy and manage complex cloud applications. This goal is achieved by requiring developers to define an abstract topology of a complex application and to create plans describing its deployment and management.
2. **Portability of application descriptions and their management** but not the actual portability of the applications themselves). To this end, TOSCA provides a standardized way to describe the topology of multi-component applications. It also addresses management portability by relying on the portability of workflow languages used to describe deployment and management plans.
3. **Interoperability and reusability of components**. TOSCA aims at describing the components of complex cloud applications in an interoperable and reusable way [13], [14] and [15]

Compared to other standards efforts (CAMP, CIMI, EMML, OCCI, Open-CSA, OVF, SOA-ML, and USDL), TOSCA focuses on all the three goals above [10] as shown in Figure 5.

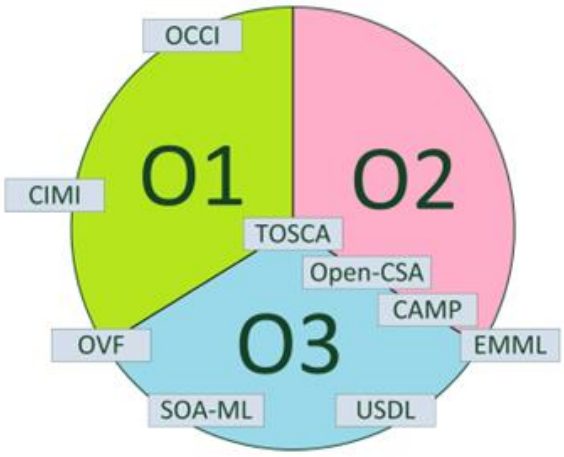

Figure 5, Comparison between TOSCA and other standard efforts

The TOSCA language specification [16] and [17] (now based on YAML [18]), allows the description of topologies, nodes and relationships (see Figure 6) at three different levels of abstractions (see Figure 7):

- Types: akin to a Java Abstract class,
- Templates: akin to a Java Concrete class
- Instances: akin to a Java instance of a class.

The combination of these three levels of abstraction supports re-usability of descriptions and offers a flexible and expressive syntax for the definition of application templates at different level of granularity and definition. Such an approach, also supports implementation where profiles can be automatically completed to ease the burden of complete specifications [11].

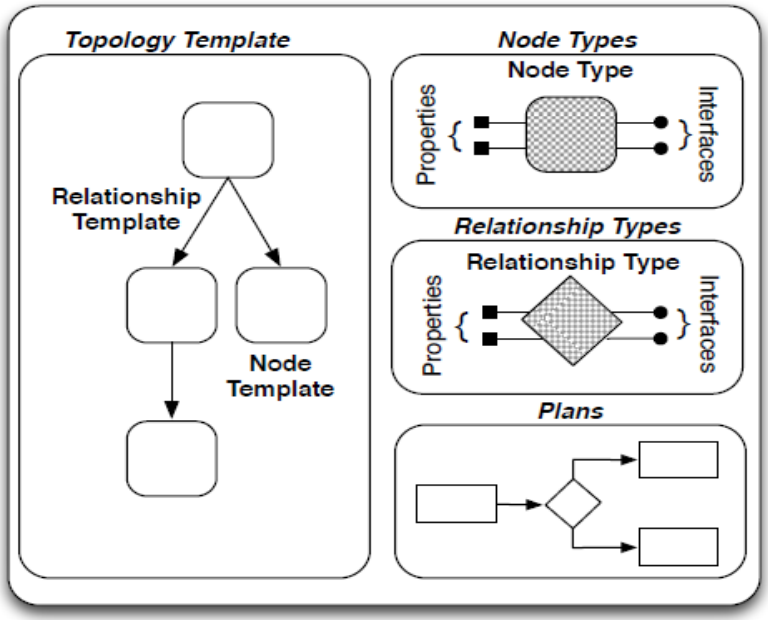

Figure 6, Topologies, nodes, relationships and plans in TOSCA

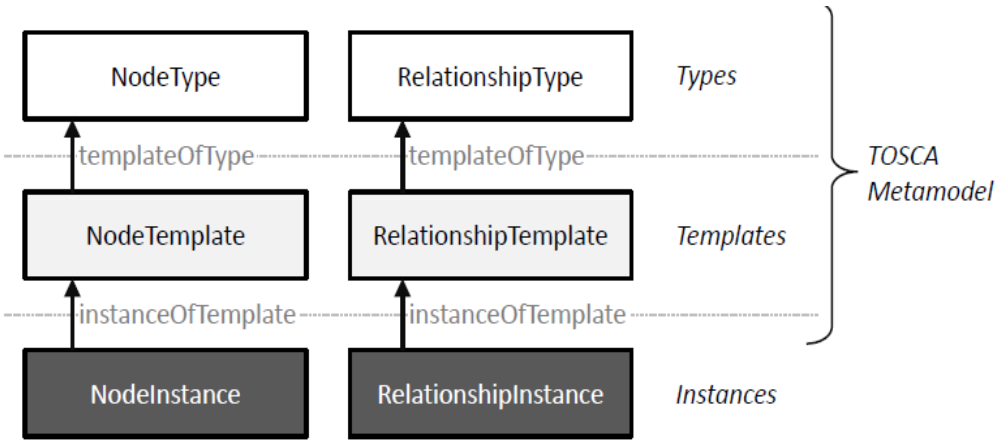

Figure 7, Types, templates and instances in TOSCA

In addition to the description of a topology (nodes and relationships), TOSCA provides for the definition of implementation plans (see Figure 8) to enact the deployment of nodes and relationships.

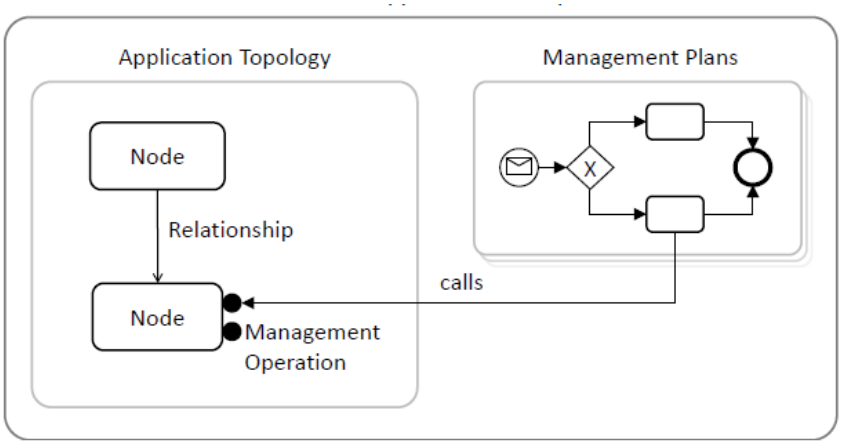

Figure 8, Topologies specification and management plans

TOSCA [10] and [16] also prescribes the format to archive application specifications along with the installable and executables needed to properly instantiate the specified applications.

The topology specification must be packaged together with the artefacts implementing its components so as to make all such artefacts available to the execution environment. The TOSCA specification [10] defines an archive format called CSAR (Cloud Service Archive) to package application specifications together with concrete implementation and deployment artefacts. A CSAR is a zip file containing at least the Definitions and TOSCA-Metadata directories. TOSCA containers can deploy applications by processing the CSAR archives in two different ways: imperative and declarative [19]. On one hand, imperative processing takes the CSAR and deploys the application according to the workflow defined as a Build Plan in the corresponding Service Template. On the other hand, declarative processing

deploys the application by trying to automatically excerpt a deployment plan from the application's Topology Template.

TOSCA does not mandate the use of any specific mechanism or technology for client authentication. However, a client must provide a principal or the principal must be obtainable by the infrastructure [7]. None of the reviewed publications on TOSCA mention support for authentication, authorization, trust and privacy. However, security and other policies can be defined using TOSCA extensions that can be delegated to other components to implement such policies.

As TOSCA is NOT a language or an implementation, multiple attempts have been tried to build TOSCA-compliant environments including TOSCA containers. The implementation of a Proof of Concept prototype [[20] with Chef and OpenStack describes the steps needed to implement a TOSCA-enabled environment and the technical difficulties in parsing the original TOSCA-XML based language. More recently, the OpenTOSCA [21] and [22] initiative, offers a TOSCA environment with the following components:

- **Winery** [23] for the creation and modelling of TOSCA applications, including graphical modelling of topologies and management plans. Exported as Cloud Service Archive (CSAR) for TOSCA runtime.
- **OpenTOSCA container** [24] to process CSARs, run plans and manages state.
- **Vinotek** [25] to hide technical details and to provide end users a simple graphical interface to provision Cloud applications on demand.

TOSCA is also related to other initiatives such as: TOSCA-MART, a method that enables deriving valid implementations for custom components from a repository of complete and validated cloud applications. TOSCA-MART enables developers to specify individual components in their application topologies, and illustrates how to match, adapt, and reuse existing (fragments of) applications to implement these components while fulfilling all their compliance requirements.

6.2.2 CAMP

CAMP [26] is a simple API specification to standardize the API of PaaS systems like ORACLE Cloud, CloudBees, OPENSHIFT, etc. The specification offers an HTTP-based RESTful API combined with JSON representation of data. It is extensible to be compliant with future changes. The specification is language- (Ruby, Java, Python, PHP, etc.), framework- (Rails, Spring, etc.), and platform- (Java EE, .Net, etc.) neutral. The CAMP API has been designed for management of applications. CAMP also contains a resource and lifecycle model for application management.

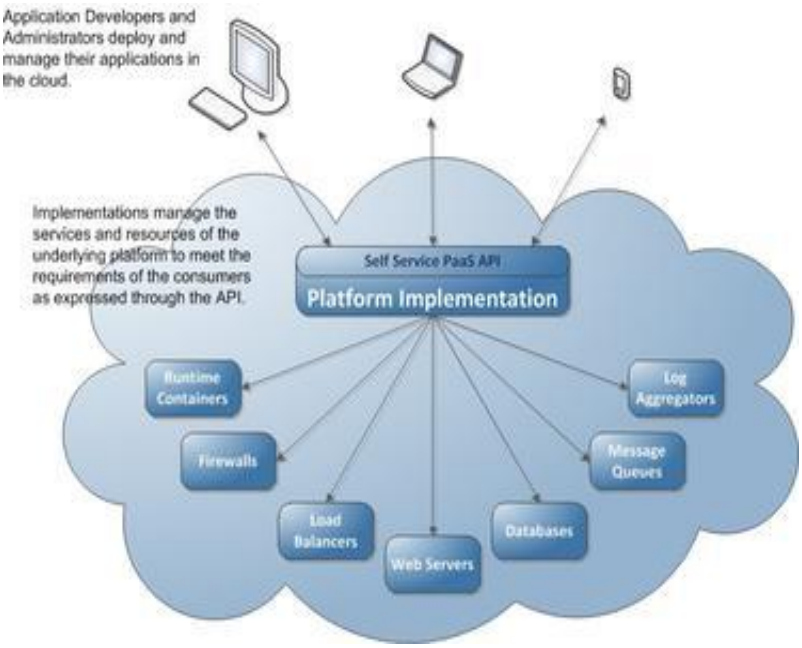

Figure 9, CAMP entities

The lifecycle model (see Figure 10) supports performing, uploading, configuring, customizing, deploying/undeploying, starting/stopping, snapshotting, suspending/restarting and deleting operations on an application/service as well as monitoring the operation of the application.

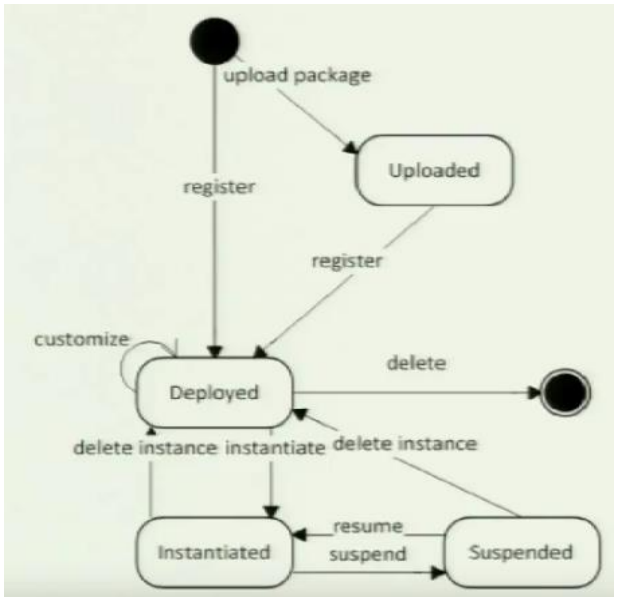

Figure 10, CAMP lifecycle

A candidate for CAMP implementation on the PaaS side is the SOLUM [27] project. Although the Solum API and resource model are similar (and in some cases identical) to the API and resource model defined in the CAMP specification, they are also different in a

number of significant ways. Tools and applications written to consume the CAMP API cannot use the Solum API.

TOSCA and CAMP are not at the same abstraction level. Their goals are different. CAMP as an API could be directly used in applications where at certain points of the applications the CAMP commands are used and the implementing PaaS system should execute these commands typically first translating the CAMP commands into their native API commands. CAMP does not deal with topology and orchestration. There is no description on how the application looks like. It defines only application template requirements and resource template capabilities. TOSCA is about to describe how to deploy and manage a complex application in the cloud and in order to achieve this goal it provides descriptors. Implementing TOSCA means to interpret these descriptors and according them to deploy and manage the application in the cloud. It typically requires a tool that can do this interpretation as well as the execution management.

The tool that interprets TOSCA commands can apply the CAMP API in its executor component and as a result TOSCA descriptors will be executable on top of PaaS systems that will implement the CAMP API. Figure 11 shows these levels of abstractions and the chain of tools and specifications that can work together at different levels of abstractions.

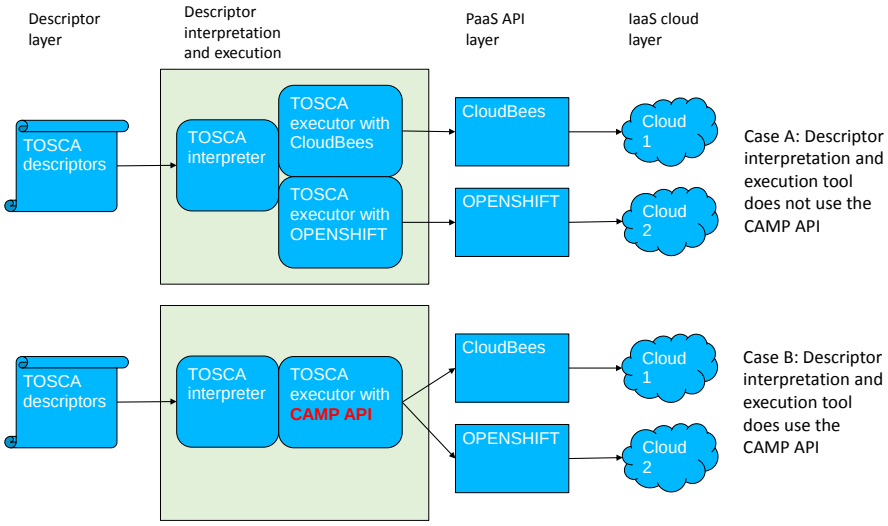

Figure 11, Topologies Specification and Management Plans

7. Cloud Orchestration Tools and Application Description Approaches

WP6 has considered several cloud orchestration tools to be used in the MiCADO platform:

- Chef,
- Heat,
- Juju,
- Occopus, and
- Puppet.

WP5 investigated how these cloud orchestration tools describe applications and services to support their deployment, execution and monitoring on the Cloud:

7.1 Application Description Approaches used by Cloud Orchestration Tools

7.1.1 Chef

Chef[28][29] is open source cloud orchestration tool that supports integration with cloud-based platforms. It launches and maintains servers, and manages clients that run on nodes, which can be physical or virtual machines. This client performs the automation tasks the specific node requires. The nodes register at a server, which then provides recipes defining these automation tasks and assigns roles. Cookbooks are used to organize related recipes, which are basically Ruby scripts, and supporting resources. Roles contain lists of recipes, which are then executed by the Chef client upon retrieval from the server leading to the desired configuration.

Chef uses a **pure-Ruby, domain-specific language (DSL)** to describe system configuration. It explicitly describes how to deploy and connect cloud application components. Chef uses "**recipes**" and "**cookbook**s. Each deployment step can be described independently, and bringing those independent recipes together creates a repeatable application deployment process. Since every operations step can be described in a recipe, there is nothing that can be deployed manually that Chef can't automate. Recipes define how infrastructure components are to be deployed and managed. Cookbooks are a collection of recipes that define scenarios like setting up a database with all dependencies considered. By abstracting and through that special naming it is emphasized that the defined recipes are supposed to be platform independent and are usable in other projects. There already exists a pre-defined library with many recipes

7.1.2 Heat

Heat[30][31] is a pattern-based orchestration mechanism developed by OpenStack. It provides a template-based orchestration for describing a cloud application by executing appropriate OpenStack API calls that generate running cloud applications. The software integrates other core components of OpenStack into a one-file template system. The templates allow for the creation of most OpenStack resource types as well as more advanced functionality such as instance high availability, instance auto-scaling, and nested stacks. These templates, called **Heat Orchestration Templates (HOT),** are native to Heat and are expressed in **YAML**. These templates consist of:

- **Resources** (mandatory fields) are the OpenStack objects that must be created, like server, volume, object storage, and network resources. These fields are required in HOT templates. Each resource consists of:
  - References—used to create nested stacks

  - Properties—input values for the resource
  - Attributes—output values for the resource
- **Parameters** (optional) denote the properties of the resources.
- **Output** (optional) denotes the output created after running the Heat template, such as the IP address of the server.

7.1.3 Juju

Juju[32][33] is an open source automatic service orchestration management tool that enables to deploying, managing, and scaling software and services on a wide variety of cloud services and servers. It can significantly reduce efforts needed for deploying and configuring a product's services. Juju utilizes **charms** to simplify deployment and management tasks. A charm is a **set of scripts** that can be written in any language. After a service is deployed, Juju can define relationships between services and expose some services to the outside world. Charms encapsulate application configurations, define how services are deployed, how they connect to other services, and how they are scaled. Charms define how services integrate, and how their service units react to events in the distributed environment, as orchestrated by Juju. Charms usually include all of the intelligence needed to scale a service horizontally by adding machines to the cluster, preserving relationships with all of the services that depend on that service. This enables developers to build and scale up and down the service on the cloud.

7.1.4 Occopus

Occopus[34] is an orchestrator tool to build network of nodes containing interconnected services forming a virtual infrastructure. Each node is built up by virtual machines executing services or application. The configuration and initial setup of nodes can be performed by contextualization and configuration management tools. The built-up infrastructure can then be continuously maintained and the health of nodes can be monitored to detect and recover faulty nodes. Occopus has a pluggable architecture to cooperate with different type of resources, with different type of configuration management (CM) tools, to apply different contextualization methods and procedures. As a result Occopus can be utilized in a broad range of environments by applying any combination of its plugins. It uses descriptors, written in YAML, to describe the services to be deployed in the Cloud. Occopus deploys the services according to deployment order specified in the descriptor. It does not only deploy the services but checks their availability and accessibility before deploying the next service. Furthermore, the descriptor can contain contextualization information for every deployable service and based on that information Occopus carries out contextualization for the deployed services. As a result after contextualization the services can reach each other, i.e. they can collaborate to realize a higher level service (the virtual infrastructure).

71.5 Puppet

Puppet[35][36] manages cloud services supporting several Linux distributions and Windows versions. It is designed to manage an infrastructure while pre-defined or custom modules are stated via relationships, following a high-level concept. A module is a graph of relationships that is reusable. The user interface is graphical, and because of the declarative language the programming effort is minimized. Necessary information is discovered during runtime and compiled into manifests. **Manifests** are the idempotent declarative Puppet programs. It uses either a **custom declarative language** or a **Ruby Domain Specific Language** to describe the configuration of system services, allowing the definition of reusable modules. A system-specific catalogue holds the resource and its dependencies and Puppet then applies any required actions to the target system, using the previously compiled manifests. Puppet supports an iterative life-cycle with four stages: Define, Simulate, Enforce and Report. In the definition stage the relationship graph with modules and their desired state is declared. It is

then possible to simulate any changes to the system before committing it, allowing testing and avoiding faulty updates.

7.1.6 Summary

The development of the above described cloud orchestration tools either has been started before TOSCA has been elaborated or has been running parallel with TOSCA development. As a result, they have their own application description solutions: Chef and Puppet uses Ruby Domain Specific Language, Heat and Occopus uses YAML, and Juju uses scripts. After publishing TOSCA they either created plug-ins to process TOSCA-based application descriptions or developed translators to convert these descriptions to their native descriptions.

IBM developed the TOSCA Chef plug-in to process TOSCA based application descriptions by **Chef**. TOSCA support can be enabled by installing and enabling a pattern type “TOSCA Foundation Pattern Type” in IBM SmartCloud Orchestrator. This orchestrator imports and deploys TOSCA service templates as virtual application patterns or virtual application templates. Before deploying an application pattern imported from a TOSCA CSAR, basic configuration for virtual application patterns, such as configuration of cloud groups or import of base images must be performed. The TOSCA Chef plug-in is configured and deployed in a Chef client.

Heat leverages TOSCA as a standard based approach for modelling cloud stacks and applications. It uses OpenStack components for deploying TOSCA cloud stacks in OpenStack cloud. OpenStack developed the OpenStack Newton is a TOSCA Parser and Heat Translator. First, it produces in-memory graph of TOSCA nodes and relationship among them. Next, it translates non-Heat (e.g. TOSCA) templates to HOT, and deploys it with Heat.

Juju was extended to be able to parse TOSCA based application descriptions. Juju topology model components can be transformed to TOSCA compliant topology model components. Both Juju and TOSCA topology models specify graphs consisting of nodes and relations between nodes to define the structure of cloud services. In TOSCA, both relations and nodes are explicitly modelled as separate topology model components. Juju specifies nodes as topology model components only. The TOSCA-Juju transformation consists of two steps. First, a TOSCA-compliant topology model component has to be generated for each Juju charm. As a result, each node that can be modelled using Juju can be modelled using TOSCA, too. However, the relations between these nodes cannot be modelled in TOSCA because the corresponding topology model components are missing. Second, to address this issue, additional TOSCA-compliant topology model components have to be generated for each relation that can be implicitly modelled using Juju.

8. Comparison of Application Description Approaches

8.1 Comparison Criteria

The decision on which approach would be ideal to meet COLA's requirements has to be based on a comprehensive and a multi-dimensional analysis to weight strengths and weaknesses of each solution. This analysis does not attempt to define the worthiness of the perused solutions per se, but only assess their capacity of supporting COLA's design and aim.

WP5 defined the comparison criteria that are recapitulated in the Table 4 – Table 7 below. They are broadly divided into:

- **Basic Properties** that cover the support for generic functionalities such as portability, scalability and possible implementation characteristics and constraints such as the packaging of installation artefacts and the availability of examples and tutorials.
- **Entities managed and storage** that cover the capacity of the various solutions to describe the individual components and the whole application, their relationships and their overall design. This category also covers the support to publish, store query and share application description profiles.
- **QoS parameters** that covers the possibility (if not explicitly the capacity) of expressing Quality of Service parameters such as elasticity, scalability, security.
- **Application Execution** that covers the support for the execution of the applications.

Application description: basic properties

	generic	platform specific	portability support	re-usability support	Scalability support	description language	packaging	examples	tutorials
CAMP	Yes	No	Yes	Yes	No	API	PDP which may contain ZIP, GZIP and TGZ	Yes	No
TOSCA	Yes	No	**Application Portability:** application description supported, not the application **Deployment portability** deployment artifact is NOT portable **Management Portability**: via workflow language	Yes	Yes	YAML	CSAR	Yes	Yes (OPEN-TOSCA)
Amazon AWS CloudFormation	None	Amazon Web Services + Amazon Cloudformation	N/A	N/A	dynamic configuration + auto scaling	JSON -> YAML		Yes	AWS CloudFormation Template
Microsoft Azure Schema	None	Microsoft Azure	N/A	template decomposition	N/A	JSON	Visual Studio format	Yes	
ORACLE Virtual Assembly Builder (OVAB)	none	ORACLE Enterprise Platform	N/A	N/A	N/A	XML	ORACLE .ova file	Yes	ORACLE Fusion tutorial

Table 4: Basic properties

Application description: Entities Managed and Storage

	application description	topology description	service description	resource description	artefact description	GUI to give descriptions	marketplace	catalogue
CAMP	Capabilities and requirements	No	No	Yes (Capacity)	Components	No	No	No
TOSCA	Topology, Node and Relationship Type or Template	Topology Type or Template	Node Instance	Not supported	Implementation and deployment artefacts	Vinotek[37] and Winery[23]	TOSCAMart[13]	yes
Amazon AWS CloudFormation	AWS CloudFormation Template	links in AWS CloudFormation Templates	custom AMI template	base + foundational AWS templates	stack template file template configuration file	command line or AWS CloudFormation Designer	AWS Marketplace	Amazon Elastic Block Store
Microsoft Azure Schema	Azure template with resource groups	links defined in resource group		ARM template	artefacts as tools to be installed in VM	command line or Azure Portal	Azure MarketPlace	Directory of Azure Cloud Services
ORACLE Virtual Assembly Builder (OVAB)	OVAB assembly	metadata in OVAB assembly	OVAB template	Appliance	several artefacts types	command line or OVAB Studio	ORACLE Cloud Marketplace	software library of Oracle VM virtual environment

Table 5: Entity managed + storage

Application description: QoS parameters

	authentication	authorization	privacy	trust	security groups	elasticity	scalability
CAMP	N/A	N/A	N/A	N/A	N/A	no	no
TOSCA	indirectly: through policies	Indirectly: through policies	Indirectly: through policies	Indirectly: through policies		ElasticTOSCA	Indirectly: through policies
Amazon AWS CloudFormation	AWS Identity and Access Management	AWS Identity and Access Management	AWS Privacy Policy	AWS Trust Policy	EC2 security groups	AWS Elastic Beanstalk	AWS Scaling Policy
Microsoft Azure Schema	Multifactor authentication	RBAC	secrets	N/A	network security groups	Windows Azure Caching	Windows Azure Caching
ORACLE Virtual Assembly Builder (OVAB)	Authentication token	security roles	OVAB Privacy Policy	OVAB Trust Policy	OVAB security groups	Oracle Exalogic Elastic Cloud	

Table 6, QoS Parameters

Application execution

	runtime support	container support
CAMP	No	No
TOSCA	Yes	Docker
Amazon AWS CloudFormation	AWS EC2	Docker + others
Microsoft Azure Schema	Windows Azure Runtime	Azure Container Service + Docker
ORACLE Virtual Assembly Builder (OVAB)	ORACLE Enterprise Manager 12C	ORACLE Application Container

Table 7: Application execution

9 Application Description in COLA

After consideration of all the previously mentioned factors, TOSCA was selected as the best candidate for the Application Description Templates in COLA. Such a decision pivots on the following findings:

- **Basic Properties**. TOSCA offers support for generic functionalities (portability, scalability, etc.) suitable for the project, it offers the packaging of installation artefacts and supports a large variety of installation methodologies that vary from simple scripts to complex workflows. Furthermore and quite importantly, TOSCA is an accepted standard, it is supported by a strong and growing community, and there is ample literature of several and successful attempts of its usage by the research community.
- **Entities and storage.** TOSCA's philosophy is very similar and highly compatible with COLA's three layered concept to describe applications, their components, their relationships and generic templates. TOSCA also supports the possibility to publish, discover and share application description templates.
- **QoS parameters.** TOSCA does not directly or explicitly support QoS parameter but it is flexible and generic enough to allow them to be described.
- **Application Execution.** TOSCA per se is a language specification so it does not directly support runtime or container support. However, there are TOSCA implementations that do so.

Greater details and references for each of the decision points are listed in Table 8.

As a further argument to select TOSCA is interoperability. Even today many cloud orchestration tools are able to manage TOSCA based application/service descriptions. (See details in Chapter 7). Their number is increasing every year. Selecting a TOSCA based approach to specify applications/services will improve shareability of COLA applications.

Criteria	Support	Notes	Ref
Generic	Yes	OASIS specification not an implementation. Implementation is OpenTOSCA	[7] [38]
Platform Specific	No		[7]
Portability	Yes (with caveat)	Description is, Application is NOT per se	[8]
Language	Yes	YAML	[17]
Packaging	Yes	CSAR	[39]
Examples	Yes	Theoretical Examples[8] Usage of TOSCA to enhance portability[40]	[8] [40]
Tutorials	Yes	OpenTOSCA has tutorials available	[41]
Application Description	Yes	Note and Relationship Templates	[8]
Topology Description	Yes	Note and Relationship Templates	[8]
Service Description	Yes	Node and Relationship Types	[8]
Node Description	Yes	Node Template, node type, node instance	[8]
Artefact Description	Yes	CSAR	[39]
GUI	Yes	Winery	[23]
Marketplace	Yes	TOSCA-Mart	[13]
Catalogue	Yes	TOSCA-Mart	[13]
Authentication	Yes	Through Policies and management plan workflows	[8]
Authorization	Yes	Through Policies and management plan workflows	[8]
Privacy	Yes	Through Policies and management plan workflows	[8]
Trust	Yes	Through Policies and management plan workflows	[8]
Elasticity	Yes	Through management plan workflows	[8]
Scalability	Yes	Through Scalability Policies	[17]
Runtime Support	Yes	Offered by Open Tosca [38]. Automatic Topology Completion[11]	[11][38]
Container Support	Yes	Docker, Rocket, Kubernetes and mesos	[42]

Table 8: TOSCA References

10 Cloud Orchestration in COLA

WP6 investigated and analysed several cloud orchestration tools as prospective candidate for the MiCADO platform. (See details in Chapter 7). WP6 selected Occopus as the cloud orchestration tool to be used in COLA. Further, we give an overview how Occopus describes applications and services.

10.1.1 Describing infrastructure and node

In order to specify a particular virtual infrastructure for Occopus, developers need to provide an infrastructure description and node definition(s).

The **infrastructure description** is the most abstract form of defining the infrastructure. It does not contain any implementation-dependent details. It contains two big sections: list of nodes and dependencies.

List of nodes contains name, type, implementation selector, scaling information and variables for each node i.e. **node definition**, however only name and type are obligatory. Type is a reference to a node definition. Node definition selector is optional and may perform selection among multiple implementations (definition) of a certain node. For example, on **Error! Reference source not found.** node B uses selector while node A does not. Scaling-related information covers the minimum and maximum number of instances a node may have during the lifetime of the infrastructure. The scaling section may include rules and policies how the actual number of instances should be calculated to support auto-scaling. The variables section contains user defined key-value pairs, these variables can be referred later in other parts of the infrastructure description or in node definitions. This is mainly used to make certain parameters in node definitions tunable from the infrastructure description.

Dependencies in the infrastructure description defines if a node requires the existence of another node before coming to live. Dependencies are defined by dependency pairs like B depends on A, which means node A must be deployed before node B (as on **Error! Reference source not found.**). The entire dependency graph must be described by dependency pairs among nodes. Each dependency pair might contain connection related information, like mapping where the value of certain attribute of a node might be assigned to an attribute of another node.

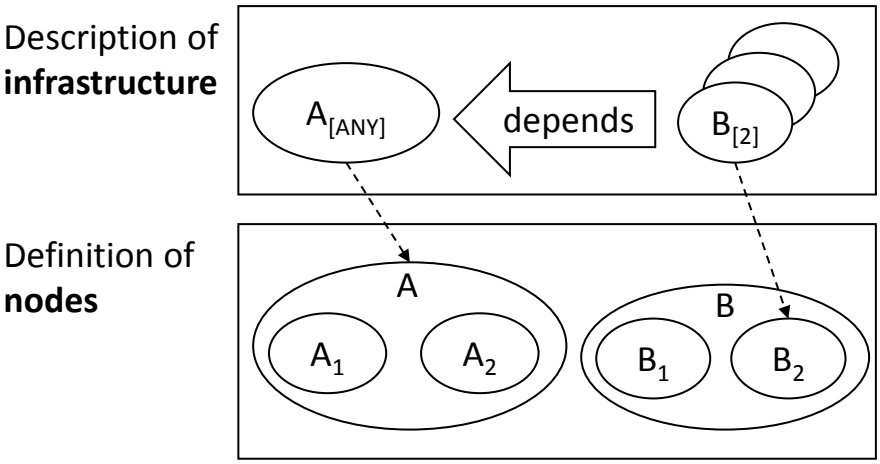

Figure 12, Relation among infrastructure description and node definition

Node definition also contains details how a node must be implemented. A node might have

more than one implementations. For example, one of the implementations can describe how the node can be deployed on an Amazon cloud, while another one defines the same for a particular Openstack cloud. The selection among implementations can be delegated to Occopus (i.e. random) or can be performed by using a selector in the infrastructure description as depicted on Figure 12.

Focusing on the implementation of a node there are one obligatory ('resource') and three optional ('contextualisation', 'config_management', 'health_check') sections to be specified (see Figure 13).

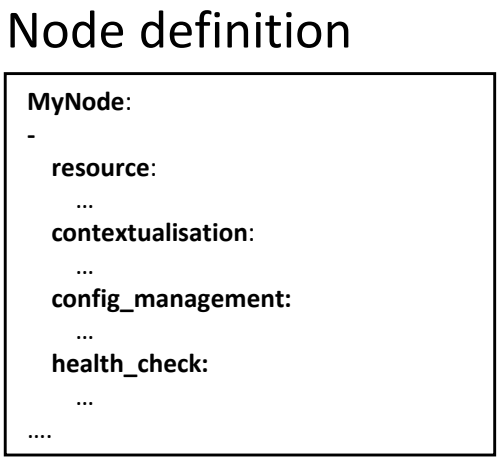

Figure 13, Sections of a node definition

10.1.2 Resource description

The **resource section** of a node definition specifies the most important and obligatory parameters to inform Occopus where and how to create a virtual machine for that particular node. In this section, there are two obligatory parameters: type and endpoint. Type informs Occopus which protocol the interface located at endpoint applies. The example on Figure 14defines an 'ec2 type of interface. The rest of the parameters in this sections are protocol specific parameters. As it can be seen, ec2 interface requires name of region, image ID and instance type to instantiate a particular virtual machine to represent a node.

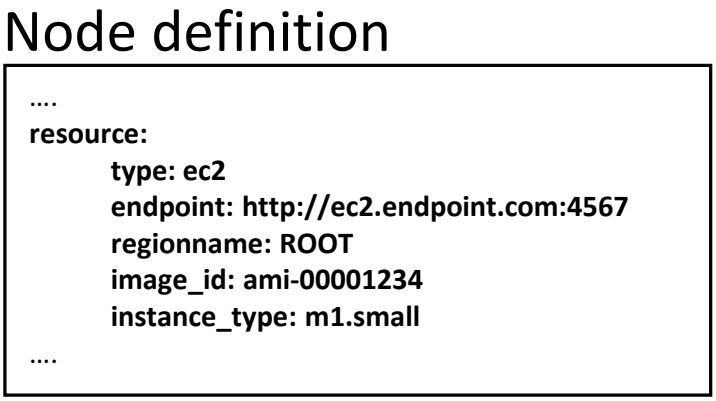

Figure 14, Resource section of a node definition

The credential information is stored in a separate authentication file for all possible interfaces and particular endpoints. Currently, ec2, nova, docker, occi and cloudbroker type resources can be used. Each of them has their own list of parameters to be defined in the resource section.

10.1.3 Contextualisation

Contextualization allows applying user defined configuration and initial settings of the newly virtual machine at start-up by executing scripts, creating config files, adding authentication keys, etc. Currently, contextualization is supported through the cloud-init tool, which is a de facto standard. In node definition one may add a complete cloud-init configuration file which is then passed as user-data when launching a new virtual machine. However, an additional extension is added for handling the contextualization information. Before passing the cloud-init configuration file as user data for example to an ec2 cloud, it is used as a Jinja2 template and it is resolved with the actual information stored in Occopus about the actual state of the infrastructure and nodes. Therefore it is possible to insert e.g. the content of a variable defined in the infrastructure description (see Figure 15), or to insert the IP address of an already launched node. The ip address can be queried from Occopus at runtime inside the contextualization with the support from Jinja2 module. With this extension contextualization information can be dynamically modified to pass information among the nodes at start-up.

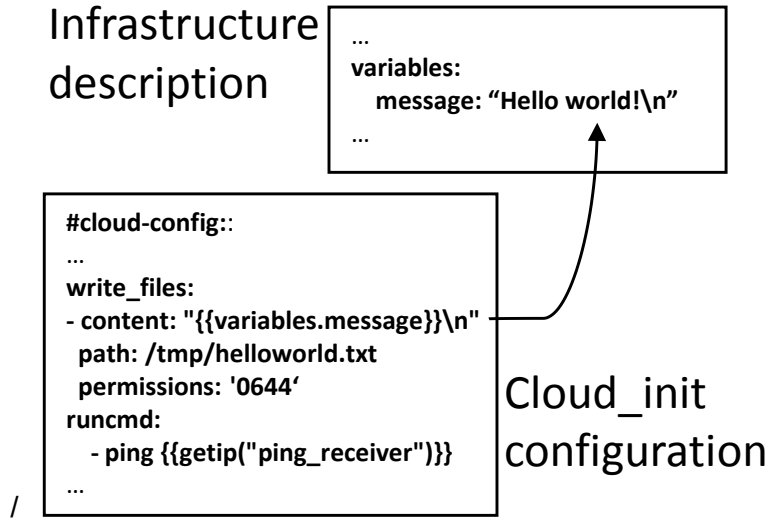

Figure 15, Example for referencing variables and ip address

10.1.4 Config management

Occopus is able to utilize external config manager tools, like Chef. In cases when nodes are built by config managers, Occopus is able to cooperate with them to organize nodes into an infrastructure. For example, in case of Chef the endpoint of Chef server and the list of recipes must be defined (see Figure 16)

Node definition

```
....
config_management:
        type: chef
        endpoint: https://chef.server.com
        run_list:
          - recipe[database-setup::db]
....
```

Figure 16, Example for config_management section in node definition

10.1.5 Health check

This section describes how Occopus should check if a particular node is still alive on that node. There are built-in methods like ping, port checking, url checking and database connectivity checking. Occopus assumes that there are healthy services on the node until all checks executed successfully. A timer is started once at least one of the checks fails and node is restarted after a timeout period – defined in the node definition – has been reached. On the example (see Figure 17), all possible checks are represented.

Node definition: health_check

```
...
MyNode:
...
health_check:
        ping: True
        urls:
          - http://{{ip}}/myapp
        mysqldbs:
          - { name: 'dbname', user: ‘dbuser', pass: 'dbpass' }
        ports:
          - 22
        timeout: 300
...
```

Figure 17, Example for health_check section in node definition

10.1.6 Occopus vs TOSCA

Occopus and TOSCA specifies applications and services in very similar way. They are based on the same concept and use similar entities::

Occopus	TOSCA
infrastructure description	topology template

node definition	node type
relation and dependency in the infrastructure description	relation type
partly supported	plan

Since the concept of Occopus and TOSCA is very similar it will be relatively easy to support TOSCA in Occopus.

11 References

[1] S. Pearce and S. Bryen, “Managing Your AWS Infrastructure at Scale,” 2015.

[2] “Learn Template Basics - AWS CloudFormation.” [Online]. Available: http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/gettingstarted.templatebasics.html. [Accessed: 20-Feb-2017].

[3] “Microsoft Azure Essentials Azure Web Apps for Developers | Microsoft Press Store.” [Online]. Available: https://www.microsoftpressstore.com/store/microsoft-azure-essentials-azure-web-apps-for-developers-9781509300594. [Accessed: 20-Feb-2017].

[4] Kai Yu, “Design and Implement a SelfService Enabled Private Cloud with Oracle Enterprise Manager 12c.” [Online]. Available: https://published-rs.lanyonevents.com/published/oracleus2015/sessionsFiles/2011/UGF9927_Yu-UGF9927_Kai_Yu_OOW.pdf. [Accessed: 20-Mar-2017].

[5] K. Yu, “SIMPLIFYING APPLICATION DEPLOYMENT IN CLOUD USING VIRTUAL ASSEMBLIES AND EM 12C.” [Online]. Available: https://kyuoracleblog.files.wordpress.com/2013/05/2013_369_yu_ppr.pdf. [Accessed: 20-Mar-2017].

[6] “Oracle Fusion Middleware.”

[7] OASIS, “OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC,” *Website*, 2014. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca. [Accessed: 15-Feb-2017].

[8] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, “TOSCA: Portable Automated Deployment and Management of Cloud Applications,” in *Advanced Web Services*, 2014, pp. 527–549.

[9] “tosca-primer-v1.0.”

[10] A. Brogi, J. Soldani, and P. Wang, “TOSCA in a nutshell: Promises and Perspectives.”

[11] P. Hirmer, U. Breitenbücher, T. Binz, and F. Leymann, “Automatic Topology Completion of TOSCA-based Cloud Applications.”

[12] C. Pahl, L. Zhang, and F. Fowley, “A Look at Cloud Architecture Interoperability through Standards.”

[13] J. Soldani, T. Binz, U. Breitenbücher, F. Leymann, and A. Brogi, “ToscaMart: A method for adapting and reusing cloud applications,” *J. Syst. Softw.*, 2016.

[14] J. Soldani, T. Binz, U. Breitenbücher, F. Leymann, and A. Brogi, “TOSCA-MART: A Method for Adapting and Reusing Cloud Applications TOSCA-MART: A Method for Adapting and Reusing Cloud Applications *,” 2015.

[15] A. Brogi and J. Soldani, “Reusing cloud-based services with TOSCA *.”

[16] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, “TOSCA: Portable Automated Deployment and Management of Cloud Applications.”

[17] W. Draft, “TOSCA Simple Profile in YAML Version 1.0,” 2014. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/csprd01/TOSCA-Simple-Profile-YAML-v1.0-csprd01.html. [Accessed: 14-Feb-2017].

[18] “The Official YAML Web Site.” [Online]. Available: http://www.yaml.org/. [Accessed: 20-Feb-2017].

[19] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, and J. Wettinger, “Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA.”

[20] G. Katsaros, M. Menzel, A. Lenk, R. Skipp, and J. Eberhardt, “Cloud Service Orchestration with TOSCA, Chef and Openstack Jannis Rake-Revelant.”

[21] T. Binz *et al.*, “OpenTOSCA – A Runtime for TOSCA-based Cloud Applications.”

[22] C. Sofokleous, N. Loulloudes, D. Trihinas, G. Pallis, and M. D. Dikaiakos, “c-Eclipse: An Open-Source Management Framework for Cloud Applications.”

[23] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann, and U. Breitenb, “Winery – A Modeling Tool for TOSCA-based Cloud Applications Title = {{Winery} --Modeling Tool for {TOSCA}-based Cloud Applications} Institute of Architecture of Application Systems Winery – A Modeling Tool for TOSCA-based Cloud Applications.”
[24] “OpenTOSCA Container – Architecture.” [Online]. Available: http://www.iaas.uni-stuttgart.de/OpenTOSCA/container_architecture.php. [Accessed: 20-Feb-2017].
[25] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann, “Vinothek – A Self-Service Portal for TOSCA.”
[26] “Cloud Application Management for Platforms Version 1.1,” 2014.
[27] “Solum CAMP API — Solum Specs 0.0.1.dev24 documentation.” [Online]. Available: https://specs.openstack.org/openstack/solum-specs/specs/juno/solum-camp-api.html. [Accessed: 20-Mar-2017].
[28] “Chef - Automate IT Infrastructure | Chef.” [Online]. Available: https://www.chef.io/chef/. [Accessed: 29-Mar-2017].
[29] M. Pfeiffer, “Chef Server on the AWS Cloud: Quick Start Reference Deployment,” 2015.
[30] R. Mateescu, “OpenStack Heat – Overview.”
[31] “Heat - OpenStack.” [Online]. Available: https://wiki.openstack.org/wiki/Heat. [Accessed: 29-Mar-2017].
[32] “Juju | Cloud | Ubuntu.” [Online]. Available: https://www.ubuntu.com/cloud/juju. [Accessed: 29-Mar-2017].
[33] J. Baker and U. S. Team, “Service Orchestration for Cloud Environments with Juju,” 2012.
[34] “Welcome - Occopus.” [Online]. Available: http://occopus.lpds.sztaki.hu/en_GB/. [Accessed: 29-Mar-2017].
[35] R. Ignazio, “Managing Mesos, Docker, and Chronos with Puppet.”
[36] “Puppet - The shortest path to better software.” [Online]. Available: https://puppet.com/. [Accessed: 29-Mar-2017].
[37] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann, “Vinothek - A self-service portal for TOSCA,” in *CEUR Workshop Proceedings*, 2014.
[38] T. Binz *et al.*, “OpenTOSCA – A Runtime for TOSCA-based Cloud Applications.”
[39] “Topology and Orchestration Specification for Cloud Applications,” 2013.
[40] C. Pahl, “IEEE CLOUD COMPUTING MAGAZINE [IN PRESS -ACCEPTED FOR PUBLICATION, 6 MAY 2015] Containerisation and the PaaS Cloud.”
[41] “OpenTOSCA Ecosystem.” [Online]. Available: http://www.opentosca.org/. [Accessed: 20-Feb-2017].
[42] C. Pahl and B. Lee, “Containers and Clusters for Edge Cloud Architectures – a Technology Review.”