A Formal and Tooled Framework for
Managing Everything as a Service

www.occiware.org

Deliverable D2.1.2

OCCIware Technical Architecture
Update

OCCIware project is supported by Fonds national pour la Société Numérique (FSN) and the following competitive clusters Systematic, Minalogic, PICOM, Images & Réseaux and Solutions Communicantes Sécurisées.
The OCCIware project aims at building a comprehensive, yet modular software engineering toolchain dedicated to service-oriented applications. In this deliverable we introduce the global technical architecture of the project. In particular, we describe the role and interfaces of each technical task.
Document versioning

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
<th>Author(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Bootstrap from D2.1.1</td>
<td>Jean Parpaillon</td>
</tr>
<tr>
<td>0.2</td>
<td>Upgrade runtime part</td>
<td>Jean Parpaillon</td>
</tr>
<tr>
<td>0.3</td>
<td>Add monitoring section 4.2</td>
<td>Amadou Diarra</td>
</tr>
<tr>
<td>0.4</td>
<td>Add contribution to monitoring section</td>
<td>Marc Dutoo</td>
</tr>
<tr>
<td>0.5</td>
<td>Add distributed deployment architecture (Section 4.3.1)</td>
<td>Maël Audren</td>
</tr>
<tr>
<td>0.6</td>
<td>Integrate dashboard section (from Nabil Djarallah)</td>
<td>Jean Parpaillon</td>
</tr>
<tr>
<td>0.7</td>
<td>Integrate DCaaS use case section (from Nabil Djarallah)</td>
<td>Jean Parpaillon</td>
</tr>
<tr>
<td>0.8</td>
<td>Integrate Monitoring infrastructure description (from Nabil Djarallah)</td>
<td>Jean Parpaillon</td>
</tr>
<tr>
<td>0.9</td>
<td>Add OCCI Generic console section 4.4</td>
<td>Jean Parpaillon</td>
</tr>
<tr>
<td>0.10</td>
<td>Add OCCIInterface section 4.5</td>
<td>Marc Dutoo</td>
</tr>
<tr>
<td>1.0</td>
<td>Review and typo corrections</td>
<td>Philippe Merle</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Marc Dutoo</td>
</tr>
</tbody>
</table>

Document reviews

<table>
<thead>
<tr>
<th>Review</th>
<th>Date</th>
<th>Ver.</th>
<th>Reviewer(s)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plan</td>
<td>March 2nd, 2016</td>
<td>0.1</td>
<td>Jean Parpaillon</td>
<td></td>
</tr>
<tr>
<td>Brouillon</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validation</td>
<td>December 14th, 2016</td>
<td>1.0</td>
<td>Marc Dutoo</td>
<td></td>
</tr>
<tr>
<td>BAT</td>
<td>December 15th, 2016</td>
<td>1.0</td>
<td>Philippe Merle</td>
<td>Ready for delivery</td>
</tr>
</tbody>
</table>
Contents

1 Introduction 9
   1.1 Objectives .............................................. 9
   1.2 Challenges ............................................. 9
   1.3 Overall Architecture .................................. 9

2 Conceptual Foundations 13
   2.1 Meta-Model .............................................. 13
   2.2 Domain Specific Languages
       2.2.1 Rationale ........................................... 14
       2.2.2 OCCI DSL - Structural Part ....................... 14
       2.2.3 OCCI DSL - Behaviour Part ....................... 15
       2.2.4 A Particular Model Specific Language: QoS ...... 16
   2.3 Models ................................................. 16

3 Eclipse Toolchain 19
   3.1 OCCI Studio ............................................. 19
   3.2 Simulator .............................................. 22
   3.3 Decision Support Tool for Cloud Services Transition .. 23
   3.4 Generators and Connectors
       3.4.1 Generators ......................................... 24
       3.4.2 Connectors ......................................... 24

4 Runtime Support 27
   4.1 OCCI Runtime kernel
       4.1.1 Objectives .......................................... 27
       4.1.2 OCCI Core implementation ......................... 27
       4.1.3 OCCI Extensions Implementation ................. 28
       4.1.4 OCCI Configurations Implementation ............. 28
       4.1.5 OCCI Backends ..................................... 28
       4.1.6 Runtime General Architecture ..................... 28
   4.2 Monitoring with OCCIware
       4.2.1 Monitoring what? Metrics and sensors list ....... 30
       4.2.2 Monitoring and OCCI contributions ............... 32
       4.2.3 Specification ....................................... 35
       4.2.4 OCCIware monitoring platform: objectives and architecture 40
   4.3 Distributed Deployment ................................ 44
       4.3.1 Architecture Schema ................................ 44
       4.3.2 Sequence diagram ................................... 44
       4.3.3 Ozwillo ............................................. 46
4.3.4 Roboconf .................................................. 46
4.3.5 Proactive Cloud Automation .............................. 47
4.3.6 Input and Output ........................................ 48
4.3.7 Scalair ....................................................... 50
4.4 OCCI Generic Console - Open API-based frontend ... 50
  4.4.1 OpenAPI OCCI Implementation ........................ 50
  4.4.2 OpenAPI frontend ........................................ 50
4.5 OCCI Generic Console - Playground .......................... 51
  4.5.1 Goal ......................................................... 51
  4.5.2 Deployment use cases ................................... 52
  4.5.3 Features .................................................. 53
  4.5.4 Illustration ................................................. 53
4.6 IaaS Administration Console .................................. 53
  4.6.1 Main IaaS Administration Console Components ........ 54

5 Use Cases .......................................................... 61
  5.1 DCaaS Use Case Description ................................ 61
    5.1.1 Elasticity System and Automatic Deployment .......... 61

6 Conclusion .......................................................... 65

A OCCI Swagger Description ........................................ 67
List of Figures

1.1 OCCIware Overall Technical Architecture ............................................. 11
2.1 OCCI Core Diagram .................................................................................... 15
2.2 OCCI Runtime Model .................................................................................. 17
3.1 The Extension Designer ............................................................................... 20
3.2 The OCCI Extension Textual Editor ............................................................. 20
3.3 Rendering on Github of an OCCI Extension generated documentation ............ 21
3.4 The Docker modeler .................................................................................... 22
4.1 OCCI Runtime Architecture ........................................................................ 29
4.2 OCCI Monitoring Extension ....................................................................... 33
4.3 OCCI Monitoring Example ......................................................................... 34
4.4 LAMP Application ...................................................................................... 35
4.5 LAMP Application on Multiple Infrastructures ............................................ 36
4.6 Use Cases to Illustrate Monitoring ............................................................... 36
4.7 Architecture of OCCIware Monitoring ......................................................... 37
4.8 Architecture of Monitoring with OCCIware Parts ...................................... 38
4.9 Monitoring Reference Architecture ............................................................ 41
4.10 Centreon Web Interface Login Page ........................................................... 42
4.11 Centreon Web Interface Menus ................................................................. 42
4.12 Centreon Web Interface Example ............................................................... 43
4.13 Runtime Architecture ............................................................................... 44
4.14 Sequence Diagram for Distributed Deployment .......................................... 45
4.15 Roboconf Architecture .............................................................................. 47
4.16 Roboconf Example .................................................................................... 48
4.17 PCA Architecture ...................................................................................... 49
4.18 Default Swagger Frontend ......................................................................... 51
4.19 Bootstrap-based Swagger Frontend .............................................................. 52
4.20 OCCInterface OCCI Playground Frontend ................................................ 54
4.21 Scalair’s IaaS Administration Console ....................................................... 55
4.22 IaaS Administration Console Menu ........................................................... 56
4.23 Remove or Resize a Widget ........................................................................ 56
4.24 Alarms Widgets ........................................................................................ 57
4.25 Virtual Machine/Physical Server Widget ................................................... 57
4.26 Associated Monitors with a Real Time Output .......................................... 58
4.27 Vulnerability Scanner Module ................................................................... 58
4.28 CPU, RAM and Disk(s) Graphs ................................................................ 59
4.29 Monitoring Widget .................................................................................... 59
Deliverable D2.1.2
Expected 2016/03/31
Delivered December 15, 2016
Status Final
Dissemination Public
Version 1.0

4.30 Other Monitoring Views .......................................................... 60

5.1 Infrastructure Architecture for Carpool Application ....................... 62
5.2 Elastic Carpool Application with ProActive and Roboconf .......... 63
## List of Tables

<table>
<thead>
<tr>
<th>Table ID</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>OCCI Concepts mapping to Metamodel hierarchy</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>List of metrics</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>List of HTTP API metrics</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>List of HTTP API computed metrics</td>
<td>32</td>
</tr>
<tr>
<td>4.5</td>
<td>OCCIware IaaS Monitoring Architecture Configurations</td>
<td>40</td>
</tr>
<tr>
<td>Deliverable</td>
<td>Status</td>
<td>Expected</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>D2.1.2</td>
<td>Final</td>
<td>2016/03/31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>December 15, 2016</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Objectives

The objective of OCCIware is to build a formal, model-driven software engineering toolchain dedicated to service oriented software, addressing cross-domain and cross-vendor engineering concerns.

Neither model-driven toolchain, service oriented software development dedicated toolchain and interoperability frameworks are innovative on their own. Model-driven software engineering is extensively used in complex systems development or critical applications and the Eclipse Modeling Framework is dedicated to that. Regarding service oriented software development, it exists a large diversity of dedicated framework like Django REST Framework [2] for Python, Jersey [4] or RESTX [6] for Java, webmachine [13] or cowboy [1] for erlang, just to cite a few. Finally, many cloud computing interoperability frameworks have been developed, addressing one or several domains (amongst IaaS, PaaS or SaaS), as libraries (jClouds [3]) or service themselves (CompatibleOne [16]).

The innovation targeted in OCCIware is the combination of these scientific and technical tools into an homogeneous toolchain.

1.2 Challenges

The integration of these different disciplines and tools, as well as their reuse by targeted audiences implies clear interfaces and pivot format to be defined.

For instance, while the Eclipse Modeling Framework can be considered as a de facto standard technology in meta-modeling communities, existing OCCI implementations does not use this framework. Hence, in order to reuse these runtimes and their associate ecosystems (connectors, developers, etc.), a data format understandable by these runtimes as well as generated by the OCCIware Studio needs to be defined and documented.

Hence, this document aims at defining the second version of the components and associated interfaces of the project. It is planned that these artefacts will evolve by the time use case developers will provide feedback regarding the architecture.

1.3 Overall Architecture

As illustrated on Figure 1.1 the different components of the project can be divided into three categories, related to the application development process steps.
THINK: these are conceptual tools allowing the description of applications independently from the implementation; they include meta-model (see Section 2.1), non-functional models (see Section 2.3) and domain specific languages (see Section 2.2);

DESIGN: design tools offer developers a way to manipulate conceptual tools graphically or textually (see Section 3.1), then to transform them into target languages: runtime configurations, documentation, simulations (see Section 3.2), decision support tools (see Section 3.3), etc.;

EXECUTE: finally, execution components implement the conceptual tools and are able to execute the designed applications.
Figure 1.1: OCCIware Overall Technical Architecture
<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Status</th>
<th>Expected</th>
<th>Dissemination</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2.1.2</td>
<td>Final</td>
<td>2016/03/31</td>
<td>Public</td>
<td>1.0</td>
</tr>
<tr>
<td>Delivered</td>
<td></td>
<td>December 15, 2016</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 2

Conceptual Foundations

2.1 Meta-Model

In software engineering, the use of models is more and more recommended in the trend of Model Driven Engineering (MDE). This should be contrasted with the classical code-based development techniques. The MDE approach is meant to increase productivity by maximizing compatibility between systems (via reuse of standardized models), simplifying the process of design (via models of recurring design patterns in the application domain), and promoting communication between individuals and teams working on the system (via a standardization of the terminology and the best practices used in the application domain). Open Cloud Computing Interface as a standard for IaaS follows this direction. A modeling paradigm for MDE is considered effective if its models make sense from the point of view of a user that is familiar with the domain, and if they can serve as a basis for implementing systems. The models are developed through extensive communication among product managers, designers, developers and users of the application domain. As the models approach completion, they enable the development of software and systems.

In MDE world, a model always conforms to a unique metamodel. One of the currently most active branch of Model Driven Engineering is the approach named model-driven architecture proposed by OMG and technically supported within the Eclipse ecosystem of programming and modelling tools (Eclipse Modeling Framework). This approach is based on the utilization of a language to write metamodels called the Meta Object Facility or MOF. Typical metamodels proposed by OMG are UML, SysML, SPEM or CWM. All the languages presented below could be defined as MOF metamodels instance of a metamodel.

In the Eclipse ecosystem of programming and modelling tools (Eclipse Modeling Framework (EMF)), Ecore can be seen as a technical implementation of essential MOF. EMF is an eMOF implementation and code generation facility for building tools and other applications based on a structured data model. From a model specification, EMF provides tools and runtime support to create Domain Specific Language (DSL) on top of the Eclipse platform. Most important, EMF provides the foundation for interoperability with other EMF-based tools and applications using a default serialization strategy based on XMI. Consequently, EMF has been used to implement a large set of tools and thus, evolved into an efficient Java implementation of a core subset of the MOF API. As a first real benefit, EMF provides a transparent compatibility of the Models infrastructure with several design environments. All the tools built with frameworks such as Xtext, EMFText, GMF or Sirius can be directly plugged in the Models infrastructure to monitor the running system. The generated code is clean and provides an embedded visitor pattern and an observer pattern. EMF also provides an XMI marshaller and unmarshaller that can be used to easily share models. Finally EMF offers lazy loadings of resources allowing the
loading of single model elements on demand and caching them softly in an application.

OCCI defined a metamodel for all kinds of management tasks without a real metamodel implementation. On top of this modeling language, OCCI also defines Protocol and API for interacting with this model. Protocol and API follows a Representational State Transfer (REST) architecture. OCCI was originally initiated to create a remote management API for IaaS model based Services, allowing the development of interoperable tools for common tasks including deployment, autonomic scaling and monitoring. It has to evolve into a flexible modeling approach with a strong focus on integration, portability, interoperability and innovation while still offering a high degree of extensibility. In OCCIware, we provide an implementation of OCCI metamodel with the Eclipse Modeling Framework. We define a set of tools to design OCCI models and OCCI extensions in preserving the API, protocol and the REST style, well accepted in industry. Based on this metamodel, we also provide a clear static semantics and operational semantics using OCL and Alloy to precisely define the OCCI domain.

2.2 Domain Specific Languages

2.2.1 Rationale

The Open Cloud Computing Interface metamodel is resource-oriented: typing mechanism involves “Kind” and “Mixin” while instantiation mechanisms are called “Resource”, “Link” and are extensible. These are, amongst others, some specificities of this metamodel with regard to, for instance, object-oriented programming. This particular paradigm justifies the need for defining Domain Specific Language(s) (DSL). DSL allows the programmers to manipulate the real concepts, and only those ones, of the paradigm they are working with. Designing a DSL for each particular class of problem is called Language Oriented Programming [12].

Another approach consists in defining General Programming Language API for manipulating the Open Cloud Computing Interface metamodel.

The DSL approach is limited by the language learning curve. While some quantitative studies exist about the complexity of solving a particular use case with a DSL vs GPL API [14], a qualitative approach requires developers feedback on multiple use cases.

In OCCIware we intend to design three DSLs, allowing to validate their usability and complexity in use cases. We consider using the API approach for some of them in order to evaluate their respective usability.

2.2.2 OCCI DSL - Structural Part

The structural part of the Open Cloud Computing Interface DSL will allow to express types and instances of Open Cloud Computing Interface concepts. The Open Cloud Computing Interface Core Specification [10] defines 8 concepts, complemented with 4 additional concepts. They are illustrated in Figure 2.1 with coloured boxes indicating additional concepts.

In order to be easily implemented, this DSL will be described with 3 notations:

Abstract Syntax: using Ecore class diagrams;

Backus-Naur Form: for direct implementation in parsers;

Graphical Notation: for use in graphical development environments (e.g. OCCIware Studio, see Section 3.1).
2.2.3 OCCI DSL - Behaviour Part

The behavioural part of OCCI DSL must allow lifecycle management of OCCI types (kinds, mixins, actions, extensions, etc), instances (resources, links, configurations, etc.) using CRUD operations. Contrary to structural language, the behavioural part can involve complex operations (e.g. arithmetic operations on attributes) and interactions between them (e.g. create or update Compute resource on application load change).

These features can be implemented in different ways described in the following sections.

“From scratch” DSL

OCCI concepts, their lifecycle as well as operations - including arithmetic, string manipulation, etc - are described with a specific language.

Xtext Based DSL

Using the Xtext tool, one can easily describe specific parts of a DSL while reusing Xtext facilities for common features (arithmetic, string or collections manipulation, etc).

Specialized API

Without being a DSL stricto sensu, an OCCI dedicated API described in a General Purpose Language (GPL) can lower the learning curve of developers.

Furthermore, several tools like GObject Introspection, or Xcore can drastically reduce the effort for synchronizing APIs in different GPLs.

---

[1] Create, Retrieve, Update, Delete

Figure 2.1: OCCI Core Diagram
2.2.4 A Particular Model Specific Language: QoS

While structural and behavioural parts of OCCI DSL will allow to manipulate OCCI metamodel concepts, the OCCIware project wants to evaluate an OCCI model-specific DSL.

As an example, a language dedicated to the manipulation of QoS resource types will be developed and will be available for use in “Datacenter as a Service” or “BigData / HPC as a Service” use cases.

2.3 Models

Models produced in OCCIware may be divided into three categories.

First, specific models will be designed to represent a particular use case, with no interoperability or standardization objective. Thanks to the Open Cloud Computing Interface meta-model and a standardized representation, these models can be interacted with from any Open Cloud Computing Interface-compliant tool, with limited semantics. These models may include:

- non-standard categories related to existing extensions, e.g. GPU-enabled compute (related task: datacenter as a service use case);
- new applications, e.g. public transportation schedule (related task: linked data as a service);
- etc.

The second category includes models aiming at being standardized, i.e. an abstraction of several tools providing a set of common features. For instance, a common model of configuration systems (Puppet, Chef, Docker, etc) should be proposed in the Deploy@OCCIware use case. Of course, specific models may be standardized, would a specific interest into that direction be demonstrated by partners and/or external contributors. But a particular effort will be put on the second category only for standardization.

Finally, wherever relevant, OCCIware components themselves will be described thanks to the Open Cloud Computing Interface formalism. It may include:

- runtime core and its components;
- extensions;
- studio and its components.

Runtime core and associated components models have already been defined as part of the Task 4.1. They are illustrated in Figure 2.2 and fully described in OCCIWARE PROJECT DELIVERABLE 4.1.1 [7].
Figure 2.2: OCCI Runtime Model
<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2.1.2</td>
<td>Final</td>
</tr>
<tr>
<td>Expected</td>
<td></td>
</tr>
<tr>
<td>2016/03/31</td>
<td></td>
</tr>
<tr>
<td>Delivered</td>
<td></td>
</tr>
<tr>
<td>December 15, 2016</td>
<td></td>
</tr>
<tr>
<td>Dissemination</td>
<td>Public</td>
</tr>
<tr>
<td>Version</td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 3

Eclipse Toolchain

3.1 OCCI Studio

The OCCI studio is a suite of tools designed to ease the development and the deployment of OCCI-based solutions. The studio implementation is based on Eclipse, which provides many frameworks for simplifying the development of OCCI studio’s tools:

- The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model.
- Sirius is an Eclipse project that eases the creation of graphical modeling workbenches by leveraging the Eclipse Modeling technologies, including EMF and GMF.
- (Eclipse) OCL is the implementation of the OMG standard language for describing constraints.
- XText is a framework for the development of DSLs with a textual syntax. It provides the parser/serializer for the syntax, along with a complete Eclipse integration (editor with syntax coloring, completion, outline).
- Acceleo is a pragmatic implementation of the OMG MOF Model to Text Language (MTL) standard. It provides a language (based on OCL) to easily describe generators based on EMF models.

The OCCI studio mainly consists in an OCCI Extension Designer (see Figure 3.1), which will help with OCCI Extensions edition. Then, for specific extensions, it will be possible to create easily specific designers.

The OCCI Extension Designer includes a graphical tool, based on Sirius, for designing the Extension using diagrams. Moreover, a textual editor allows to edit OCCI models (both extensions and configurations). Designed with XText, this utility will also provide a way to import/export textual OCCI definitions. A preliminary version of the user interface is illustrated in Figure 3.2.

An Acceleo generator can produce a documentation of the Extension encoded using the Textile markup language. Such a documentation can be rendered on a website such as github, as illustrated in Figure 3.3.

Once an Extension is described, it can be declined into a specific tooling stub that can be prepared by code generation. This tooling stub consists in:

1. an EMF metamodel project, inheriting the OCCI metamodel, and providing all the concepts of the extensions as EMF types: in this way the use of the extension with EMF is eased (modelers, generators...);
Figure 3.1: The Extension Designer

Figure 3.2: The OCCI Extension Textual Editor
Figure 3.3: Rendering on Github of an OCCI Extension generated documentation
2. A basic Sirius designer intended to be customized by the specifier, but providing a usable basic diagram editor.

A functional example of a fully developed extension is available: the Docker designer, generated from the OCCI Docker extension, which allows to create diagrams of Docker resources layout and execute them thanks to a connector integrated with the modeler. Several configurations generators are included in the Docker designer:

- docker configuration (yaml);
- curl file deploying the extension.

Figure 3.4 displays a screenshot of this application.

Figure 3.4: The Docker modeler

### 3.2 Simulator

The purpose of a simulation is to gain understanding on a system without manipulating the real system, either because it is not yet defined or available, or because it cannot be exercised directly due to cost, time, resource or risk constraints. Simulation is thus performed on a model of the system.

A simulation is generally defined through three steps. The first one consists in generating a representation of the workload, i.e. the set of inputs, to be applied to the studied system. This representation may be a trace (or scenario) depicting a real workload or a synthetic workload artificially generated by heuristics or stochastic functions. The second step concerns the simulation that is performed by applying a workload on a model of the system and producing results. Finally, the third step consists in analyzing simulation results in order to acquire a better understanding of the considered system.

These three steps may be separated and supported by different tools, like a scenario builder (as a workload generator), a model execution engine and result analysis tools. It is also possible to combine in the same tool two of these steps or even the three of them. For instance, it
It is possible to interactively create a trace while the model is executed. It is also possible to couple the execution engine and analysis tools to present, "on the fly" (i.e. during a simulation), synthetic appropriate results.

In OCCIware, we mainly use an OcciModel to model the first step. We provide an interpreter for OcciModel designed using the Gemoc approach\(^1\) that allows to bridge the graphical editors on the simulation for implementing the second step. Finally, the constraint checker (alloy and OCL) can be plugged on demand during the simulation to ensure that no constraint are violated. A full specification of the simulator is provided in D3.4.2. The constraint checker is used for the third step of the simulation.

### 3.3 Decision Support Tool for Cloud Services Transition

This tool helps to analyze the impact of the transition of an existing system to a cloud-based solution, in terms of cost. To achieve this analysis, the process is the following:

1. reverse-engineering of an existing system, as much automatically as possible but also manually, depending on the system. For instance a system that already use cloud-based solutions can be partially analyzed using OCCI connectors, that will produce an OCCI model of the system. The goal of this step is to establish a structured model of the current system, the "as is";

2. definition of a “to be”, a model at the same level of abstraction as the “as is” but specifying the system as it should be after transition. As our main use case is based on cloud only, it is entirely defined in OCCI;

3. computation of the transition cost, by adding several aspects: difference between the static cost of the “as is” and the “to be”, for instance hosting a data center vs delegating to a cloud company; effective transition cost, as in what should be updated, abandoned, created to support this transition.

To achieve this goal the tool can be based on an Obeo product: SmartEA. Based on the TOGAF framework and on open source technologies, Obeo SmartEA is designed to integrate existing repositories and develop business transformation trajectories. The data is stored using EMF CDO, which allows to use the SmartEA repository in a collaborative way. The use of a viewpoint-based approach to build models of current and future enterprise architectures greatly facilitates the execution of gap and impact analyses. These viewpoints are realized using Sirius.

SmartEA can be customized to embed the OCCI Studio tools and modelers, as they are based on the same technologies. In this way they can be used to ease the description of “as is” and “to be”.

### 3.4 Generators and Connectors

Generators and connectors enable OCCIWare models to reach out beyond their Eclipse-based MDE platform and vice versa. Therefore, they are a critical part of making OCCIWare tooling actually useful, both by integrating with other formats (generators) and runtime components (connectors).

This section describes their basic principles and provides a few examples. It will be enriched in future iterations, once use case requirements will have been gathered.

\(^1\)http://www.gemoc.org
3.4.1 Generators

Generators take advantage of the Acceleo templates component of the OCCIware tooling MDE chain to export OCCI models to other formats.

Here are the main kinds of such export targets:

1. **documentation**, be it office documents (PDF, Latex) or online (HTML) or both (ex. Markdown), to be published or used within development tools. This includes providing meta documentation such as a table of content, indexes, including referenced models that increase overall usability, and possibly even making examples executable;

2. **standard formats**, starting with the various OCCI representations, but also other cloud and architecture standards (such as UML, architecture languages),

3. and beyond that, **configuration formats** of various runtime components: erocci, Roboconf and other deployment tools, Administration Console dashboard widgets, OASIS/Ozwillo’s both Linked Data model definition and storage, etc.;

4. **code**, such as command line HTTP requests to REST APIs (curl), libraries (Java, but also scripting such as javascript - browser- or server-side, Python, Ruby, erlang) for either client (wrapper, test driver) or server (mocking and simulation, proxy), or even SNMP. Such code can constitute up to a full-fledged connector (see below), or be used in one.

Besides being used on their own to create generators, Acceleo templates are also used within connectors to generate parts of configuration applied or messages sent to achieve their purpose.

3.4.2 Connectors

Connectors provide the connection between models and running systems. Basically, connectors take a source model defined by a metamodel and project it into the running systems. Conversely, connectors monitor the running systems and transform them into the target models.

The connectors are based on the following principles:

1. **Transformation**: the connectors provide expressive model transformation techniques based on design patterns, which ease the specification of translations between the models and running systems.

2. **Introspection**: to introspect the running system, the connectors employ Model-Driven Engineering (MDE) techniques, which handle the introspection and analysis of the system at higher level of models. Using MDE techniques, different models describing certain constraints are derived and maintained at runtime.

3. **Synchronisation**: the connectors provide incremental synchronization between a running system and models. To detect model modifications efficiently, the connectors rely on a notification mechanism that reports when source model element has been changed. To synchronize the changes of model with the running system, the connectors check if model elements are still consistent by navigating efficiently between the source model and running system model using the correspondence model.

Regarding connectors targeted at the OCCI runtime, they should implement the backend interface defined in [7].

The main connectors are:
• **docker-connector**: this connector is specific to the Docker model. In Figure 3.1, the Graphical Modeler provides a tool named Docker Modeler that is used to represent the containers and the machines which host those containers graphically. This tool allows to perform some actions such as: Start, Stop, Restart, Import and Synchronize, which are reflected in the running system using the connector.

• **infrastructure-connector**: this connector is specific to the infrastructure model. This connector interacts with the most existing hypervisors and the cloud providers. Like the Docker Modeler, the Infrastructure Modeler is used the represent machines and the devices of those machine.

• **technical-architecture-analysis-connector**: to enable analysis of an Information System’s existing technical architecture as planned in task 3.5, connectors that extract models out of it are required, as defined in [11].
<table>
<thead>
<tr>
<th>Deliverable</th>
<th>D2.1.2</th>
<th>Status</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected</td>
<td>2016/03/31</td>
<td>Dissemination</td>
<td>Public</td>
</tr>
<tr>
<td>Delivered</td>
<td>December 15, 2016</td>
<td>Version</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Chapter 4

Runtime Support

4.1 OCCI Runtime kernel

4.1.1 Objectives

The objective of the OCCIware runtime kernel is to provide an OCCI-compatible API framework. Thanks to the use of model-driven techniques, it must allow developers to concentrate only on the semantic part of the API, providing generic components for handling protocols, security, rendering and all non-functional aspects of such framework.

OCCI specify concepts for describing models at different levels of the metamodel hierarchy [15][5]. The mapping between these layers and OCCI concepts is presented on Table 4.1.

<table>
<thead>
<tr>
<th>Metamodel</th>
<th>OCCI</th>
<th>Artefacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3</td>
<td>OCCI Core - Section 4.1.2</td>
<td>Core concepts implementation (e.g. Java, erlang, Ruby, Python, etc)</td>
</tr>
<tr>
<td>M2</td>
<td>OCCI Extensions - Section 4.1.3</td>
<td>DSL (e.g. XML or JSON schemas)</td>
</tr>
<tr>
<td>M1</td>
<td>OCCI Configurations - Section 4.1.4</td>
<td>REST API calls (e.g. HTTP, CoAP, XMPP)</td>
</tr>
<tr>
<td>M0</td>
<td>Backends - Section 4.1.5</td>
<td>OCCI ⇔ other API bridges</td>
</tr>
</tbody>
</table>

Table 4.1: OCCI Concepts mapping to Metamodel hierarchy

In the following sections, we describe how the runtime will implement these layers, and how each of them benefit from the outputs from SP2 and SP3.

4.1.2 OCCI Core implementation

All components of the runtime must manipulate core concepts of OCCI. For reasons detailed in [7], some components can be developed in different languages, and the core runtime provides IDL-based API for plugging these components.

OCCI Core must be implemented in all languages used for these components. Meta-model being based on EMF, the implementation of OCCI Core in Java is provided for free thanks to the EMF toolchain.

For other languages, one solution is to write code generators from EMF to the target language. As of today, only Ruby language is able to use EMF metamodels thanks to the RGen project[1]. Writing generators from EMF to a specific target language is a hard task, especially considering that the OCCI Core implementation is a critical part of the runtime, in term of performance.

Given the OCCI Core model is made up of around 10 concepts which are evolving really slowly (6 years between specifications update), it is far more efficient to implement Core model by hand rather than write a code generator for each target language.

4.1.3 OCCI Extensions Implementation

OCCI Extensions describe entities (resources or links) categories which can be instantiated by user APIs. Extensions are described through various DSL\(^2\), specified in Task 2.3 (see Section 2.2).

These DSL are independent from the runtime language, compact and focused on OCCI Core concepts. Hence, they can be easily manipulated by developers. The description of extensions in the runtime kernel is then done through these DSLs.

Thanks to the EMF toolchain, extension descriptions can be generated from graphical or textual editors, and provides powerful constraint checking, helping developers to generate correct extensions.

4.1.4 OCCI Configurations Implementation

OCCI Configurations are instances of the extensions. They are manipulated through REST API, specified by OCCI.

The runtime kernel is responsible for exposing this REST API to the users. While HTTP protocol is actually the only one fully specified, the runtime kernel architecture allows for plugging other protocols. For instance, XMPP protocol has an experimental support.

Thanks to the EMF toolchain, OCCI Configuration can be checked and validated before REST client requests are generated.

4.1.5 OCCI Backends

OCCI Backends is the most specific part of the runtime. Their are responsible for mapping OCCI concepts on existing API: existing REST APIs, databases, language-dependent APIs, etc.

As such, it is difficult to generate them but, thanks to the other components of the runtime, their development should be reduced to the really specific part.

In particular, as other components of the runtime, backends code must implement OCCI Core concepts, which can be shared with runtime kernel.

4.1.6 Runtime General Architecture

The runtime architecture is described in detail in OCCIware Project Deliverable 4.1.1 \[7\].

It is illustrated in Figure 4.1.

4.2 Monitoring with OCCIware

This section describes monitoring in OCCIware using OCCI specifications. Monitoring in OCCIware covers several aspects. First, everybody needs monitoring. This is why in a first step, we list our main requirements about monitoring: which probes, which critical parts, etc. This is about our requirements and this part is quite usual, if not short.

However, it is not in the scope of OCCIware to develop a new monitoring tool. There are already many tools and we would have few chances to make a new one successful. Then, the

\(^2\)Domain Specific Language
Figure 4.1: OCCI Runtime Architecture
main goal is to study what OCCI can bring to monitoring and design and develop this bridge between OCCI and the supervision world. This is what the two last sub-sections are mostly about.

4.2.1 Monitoring what? Metrics and sensors list

System level

Most metrics proposed below can be gathered using standard or common system administration tools, including (but not limited to) ping, top, or iostat. These metrics in Table 4.2 will be very useful to manage cloud elasticity: for example, to add or remove instances (scale out/scale in) according to system load. In addition to system metrics, there can be application and even business ones. These metrics thus depend on the monitored applications and business requirements. Use cases (SP5) will list them separately.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uptime</td>
<td>Duration since last reboot</td>
</tr>
<tr>
<td>CpuUsage</td>
<td>% usage of CPU</td>
</tr>
<tr>
<td>LoadAverage</td>
<td>% load average (last minute)</td>
</tr>
<tr>
<td>LoadAverage5</td>
<td>% load average (last 5 minutes)</td>
</tr>
<tr>
<td>LoadAverage15</td>
<td>% load average (last 15 minutes)</td>
</tr>
<tr>
<td>TotalTasks</td>
<td>% Total number of tasks (processes)</td>
</tr>
<tr>
<td>RunningTasks</td>
<td>% Total number of running tasks</td>
</tr>
<tr>
<td>SleepingTasks</td>
<td>% Total number of sleeping tasks</td>
</tr>
<tr>
<td>MemoryUsage</td>
<td>% memory used</td>
</tr>
<tr>
<td>DiskPercentUse</td>
<td>% disk occupied</td>
</tr>
<tr>
<td>DiskReadBytes</td>
<td>% Total disk read bytes</td>
</tr>
<tr>
<td>DiskReadOps</td>
<td>% Disk bytes read by second</td>
</tr>
<tr>
<td>DiskWriteBytes</td>
<td>% Total disk bytes written</td>
</tr>
<tr>
<td>DiskWriteOps</td>
<td>% Disk bytes written by second</td>
</tr>
<tr>
<td>PingStatus</td>
<td>% Network ping check</td>
</tr>
<tr>
<td>PingAvgDelay</td>
<td>% Average network round-trip delay</td>
</tr>
<tr>
<td>PingPacketLoss</td>
<td>% Network packet loss</td>
</tr>
<tr>
<td>NetworkIn</td>
<td>Number of bytes received on all network interfaces</td>
</tr>
<tr>
<td>NetworkOut</td>
<td>Number of bytes sent out on all network interfaces</td>
</tr>
<tr>
<td>ResponseTime</td>
<td>Time elapsed between a request submission and the reception of a response</td>
</tr>
<tr>
<td>Throughput</td>
<td>Number of requests processed during a time unit</td>
</tr>
</tbody>
</table>

Table 4.2: List of metrics

Application and component-level

Monitoring applications and components using system metrics All those metrics are system metrics, meaning they are about the operating system, the processes it manages and the operating system resources (CPU, RAM, disk, open files, I/O...) they consume. They are made available by the operating system. All of this means that they are both universally available and universally meaningful. In other words, whatever the SaaS application or PaaS component and whatever the IaaS infrastructure resources they run on, those metrics can be measured and will provide useful information about its current health (which, by the way, is why all monitoring solutions support them out of the box). For instance, monitoring a web application’s memory
or a database’s disk usage is mandatory to check that they are in good health. New container
technologies such as Docker make them even more useful, because they simulate an operating
system per OS process, and therefore provide the full set of metrics over whole operating systems
to each SaaS applications or PaaS component.

Monitoring applications and components beyond system metrics However, though
mandatory, system metrics are not enough for SaaS applications and PaaS components. These
have a whole range of important internal metrics, going from the most generic to the most
specific, that may have to be monitored - meaning that if said metrics are not already exposed
by them out of the box (usually in a configurable way for the most generic ones), they have
to be extracted by developing application- or component-specific sensors (unavoidable for more
specific metrics). For instance, again going from the most generic to the most specific, for a web
application, monitoring the count of HTTP requests, worker threads for a Java application, the
count of users that are logged or active in the last minute, users that are the biggest consumers
for each OS resource, or the amount of time taken by some specific critical subservices of the
application (e.g. background tasks such as rendering PDF documents out of templates), or only
for a given category of users or of business data (that’s business monitoring), are quite common
crucial metrics. For a database, monitoring the amount of operations per minute, writes not yet
written to disk, the time taken by the longest query in the last minute, or only among those
done on a given data table or collection is quite commonly crucial as well.

So while generic metrics are often exposed by its producing system / component / application
and there often already exist monitoring sensors reading them, specific metrics have first to be
extracted, and then exposed in an appropriate way, by writing custom code for said component
or application. Business applications being more specific than technical components, they are
the most in need of such custom monitoring-oriented developments.

Strategies for monitoring applications But there are good news for monitoring applica-
tions. First, whatever they run on, it often provides its own very useful monitored information.
This is not only the case for IaaS resources they are deployed on, but also application develop-
ment frameworks. For instance, the Java virtual machine exposes through its JMX monitoring
protocol information about its own threads, memory use and garbage collection state. The
Spring Boot Java framework[3] further adds to it the count of HTTP sessions and exposes it all
through HTTP. Secondly, breaking up the application’s architecture in independent subcom-
ponents as small as required allow to monitor them independently using available system- or
component-available sensors. This is microservice architecture, and container technology such
as Docker make it even more useful. Third, if a component still has no technical sensor, or if
a business-specific sensor is needed, there often exist building bricks that are available for its
development technology and make it easier to develop sensors. For instance, the Dropwizard
Metrics monitoring library provides to Java application developers useful concepts, algorithms
and glue to develop sensors and expose, aggregate and publish their monitored data.

Monitoring HTTP APIs in the Java / CXF stack This is very useful, because HTTP
APIs is probably the most common way of Cloud application to interact and expose their abilities
- and Java and CXF are sound and quite common technological choices for Cloud applications.

The aforementioned Dropwizard Metrics library, having to be used by custom code to monitor
anything, cannot be modeled and configured in OCCI itself. But it has been integrated by the
Apache CXF Java web service engine to monitor its HTTP requests, which consists in defining

³http://kielczewski.eu/2015/01/application-metrics-with-spring-boot-actuator/
and measuring a complete set of generic HTTP request metrics and their aggregation. For instance, this allows to monitor usage of HTTP APIs, such as the REST API of the Ozwillo Datacore Linked Data server’s (http://www.ozwillo.org), but also SOAP web services.

Here are this HTTP API sensor’s metrics:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Totals</td>
<td>total amount of data that has gone through the HTTP API</td>
</tr>
<tr>
<td>DataRead</td>
<td>amount of data returned by the HTTP API</td>
</tr>
<tr>
<td>DataWritten</td>
<td>amount of data sent to the HTTP API</td>
</tr>
<tr>
<td>InFlight</td>
<td>read but not yet returned and received but not yet written</td>
</tr>
<tr>
<td>CheckedApplicationFaults</td>
<td></td>
</tr>
<tr>
<td>UncheckedApplicationFaults</td>
<td></td>
</tr>
<tr>
<td>LogicalRuntimeFaults</td>
<td></td>
</tr>
<tr>
<td>RuntimeFaults</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: List of HTTP API metrics

Once this sensor has been configured for a given REST API or SOAP web service, those metrics are measured for the whole of it but also each of its operations or methods.

As a bonus, this sensor computes additional metrics out of those (thanks to Dropwizard Metrics’ "timers" providing "histogram" in addition to "meter" features), more precisely for each of them:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>time</td>
</tr>
<tr>
<td>count</td>
<td></td>
</tr>
<tr>
<td>mean_rate</td>
<td>overall mean rate since restart</td>
</tr>
<tr>
<td>m1_rate</td>
<td>mean rate over the last minute</td>
</tr>
<tr>
<td>m5_rate</td>
<td>mean rate over the last 5 minutes</td>
</tr>
<tr>
<td>m15_rate</td>
<td>mean rate over the last 15 minutes</td>
</tr>
</tbody>
</table>

Table 4.4: List of HTTP API computed metrics

4.2.2 Monitoring and OCCI contributions

Brief Reminder

Monitoring is about checking servers and software components. It can be verifying they run correctly and/or collect information (metrics).

Monitoring is an area with its own practices. These practices have been established for quite a long time now. Tools like Nagios have been there for almost 20 years and are still heavily used nowadays. Hopefully, it does not prevent new solutions from emerging, but it is hard and somehow vain to change minds.

Indeed, in the industry, people in charge of supervising systems tend to use very-well-known solutions. They rarely change their tools. As an example, although Linagora edits several software solutions, monitoring tools are almost always the same: SNMP agents, Nagios, JMX sometimes. To be clear, it is impossible to force these teams to change their tools and habits. Their position may be crucial for business and they are the closest ones from production issues. Reasons for "conservatism" in monitoring domain also include:
1. Criticism: monitoring provides alarms, etc. This can have a major impact on financial income.

2. Monitoring uses embedded software (SNMP...) which can not be replaced easily. Technically, monitoring is about probes (or sensors) that gather metrics. Then, there are collectors. Collectors can work by polling probes or by sending data to a remote location. Eventually, there are analyzing and reporting tools. Exact tools and communication protocols may vary, but when exploring this domain, one will quickly find similar solutions.

**OCCI Monitoring: purpose**

A specification was written by the OCCI working group (OGF) about monitoring. OCCI Monitoring is about providing an API to configure the monitoring of applications. So, this specification is about managing monitoring tools and their configurations, not about monitoring applications and systems.

Currently, if one wants to enable monitoring on Amazon Web Services, he or she has to use AWS’s dashboard and/or its REST API. If in addition he/she wants to collect metrics about a particular program running on VMs, it is necessary to install a different tool on this machine (such as Nagios). Monitoring data that come from AWS and those that come Nagios will each one live in their own spaces, unless you create an aggregating solution.

So, considering the example of Figure 4.5, this API could configure the monitoring at the IaaS level, the monitoring at the VM and applications level. Configuring the supervision means specifying the probes, metrics and gathering options. OCCI Monitoring does not mean we can query directly probes through such an API. It is not the spirit of this specification. It does not say anything either about exploitation tools or protocols. This is part of the implementation of OCCI Monitoring. Potentially, any monitoring tool can integrate directly or indirectly, and non-invasively, with OCCI Monitoring. The main focus of this specification is to provide a single access point to configure the monitoring solutions.

**OCCI Monitoring: model**

According to the OCCI specifications (OCCI working group, OGF), the monitoring extension requires two kinds: on one side, sensor resources process metrics and on the other side, collector Links harvest metrics from resources. The figure 4.2 shows the relation between these kinds and OCCI-core model. Indeed, the sensor kind inherits from resource and collector inherits from link.

![Figure 4.2: OCCI Monitoring Extension](image)

The sensor kind has for attributes the period of measurements(required), start time and stop time. The attributes related to the collector kind are the period of metrics gathering -
from resources) and the accuracy of sampling. To add features to these kinds, some mixins are introduced into the OCCI Monitoring extension. Thus, two mixins (Aggregator, Publisher) are associated to sensor and one to collector (the Metric mixin). These mixins are tags without attributes.

If the specification exists, there is no real example on the web of a tool using OCCI Monitoring.

### OCCI Monitoring: example

Let consider a VM on IaaS#2 in the LAMP application described above. We want to know the CPU consumption of this VM using OCCI-monitoring. So we proceed as follow:

1. Instantiate a Sensor Resource and a Collector Link connecting VM to the sensor.
2. Create a mix-in to measure CPU consumption and associate it to a collector link.
3. Create a mix-in to handle gathered values and associate it to a sensor resource.
4. Create a mix-in to publish the result and associate it to a sensor resource.

This example is illustrated in Figure 4.3.

![Figure 4.3: OCCI Monitoring Example](image)

This part is somehow very theoric. OCCI Monitoring is a good idea in the principle. But there are no open source tool that allows to configure several monitoring solutions. But such a requirement is very specific, although it is more and more important with the arising of cloud computing.

Besides, on a practical level, what does “creating a mix-in” mean? Should we be an OCCI expert to use such a tool? Let’s dig a little deeper with an example.

### Recent challenge: cloud monitoring

Monitoring is a well-established domain. However, recent technological revolutions raised new issues. It is the case with cloud computing.

- **Scalability**: in order to take into account this feature, the cloud monitoring system needs to keep monitoring efficient with potentially large number of probes.
- **Elasticity**: to support elasticity, the monitoring system needs to track virtual resources created/destroyed by expanding/contracting a cloud and to correctly handle expansion/retraction of a system.
- **Autonomy**: enabling autonomy in a cloud monitoring system is complex, since it requires the ability to receive and manage inputs from plethora of probes.
• Genericity: genericity is the ability of monitoring systems to support heterogenous resources and information. Therefore the monitoring system must have the ability to retrieve updated state from different types of resources.

Let’s illustrate these problems with an example depicted in Figure 4.4. We here consider a LAMP application, made up of a load balancer (Apache web server), an application server (Tomcat), an application (deployed in Tomcat) and a database (MySQL).

![Figure 4.4: LAMP Application](image)

For such an application, several deployment topologies are possible. This topology may be upgraded with time (as an example to face a load peak / burst load).

Each level can be replicated, so the system adapts to traffic or gets more robust: one or more Apache server(s) can load-balance traffic on one or more Tomcat(s). Some applications may also replicate MySQL databases or use a different one for each part of the application.

The number of Apache, Tomcat or MySQL nodes may vary, and can be deployed on multiple infrastructures (like IaaS). Some nodes, collocated on the same IaaS, may have private IP addresses.

On the following diagram depicted in Figure 4.5, the topology involves several cloud infrastructures. Some VMs are not publicly accessible. How one would monitor them?

• The different cloud infrastructures (IaaS 1, IaaS 2, etc.).
• The different application servers. How to deal with network restrictions?
• The deployed applications, etc.

Very few tools support these use cases, in particular the hot-upgrade of configuration files. Nagios is definitely out. Centreon, Netdata and Shinken made some steps towards this feature.

No matter what OCCIware results in, these requirements should be somehow addressed.

4.2.3 Specification

This section acts a specification for a potential tool of OCCIware, with OCCI Monitoring at its heart.

Use cases

This tool, named TINOM for This Is Not OCCI Monitoring, is about managing monitoring configurations, no matter the tools being used by the teams. By managing, we mean: listing, creating, updating, deleting, ordering and migrating monitoring configurations. This tool should be able to manage Nagios or Centreon configurations, as well as IaaS monitoring features.

We can summer up these features as the following use cases depicted in Figure 4.6. There are 2 actors shown on this diagram.
Figure 4.5: LAMP Application on Multiple Infrastructures

Figure 4.6: Use Cases to Illustrate Monitoring
Lazy is an experimented system administrator who has quite a lot of systems to monitor. These systems do not all use the same tools (legacy applications, merged project teams, incremented information systems after business acquisition, etc.) and this results in Lazy being very busy by all the things he has to manage. This tool will help him by having a single application to rule them all.

Noob is a junior in system supervision. He has no experience with usual tools and would like it to be as simple as possible while being compatible with standard approaches in system administration. This tool will help him in doing his job quickly and easily.

The use cases by themselves are quite explicit. However, let’s give a definition of a configuration. A configuration is a set of parameters (to be) applied to a given monitoring tool. These parameters may have been propagated into files (e.g. Nagios) or to APIs (e.g. Amazon’s monitoring API - CloudWatch).

**Notice**

Following a quick search on the web, it seems there is no open source tool that addresses such requirements. In fact, it is most likely that nobody is expecting such a tool. It does not mean it could not meet some success.

**Architecture**

We have very different kinds of users. Noob will definitely prefer a web interface (more intuitive and appealing for beginners). Lazy, as an administrator system, may opt in for a command-line solution. Besides, most of the use cases described previously come from OCCI Monitoring.

We can combine all of this into a single solution, which is based on a REST API depicted in Figure 4.7.

---

![Architecture of OCCIware Monitoring](image)

Figure 4.7: Architecture of OCCIware Monitoring

We can go deeper by describing how this tool interacts with other OCCIware parts. The implementation by itself should be agnostic in terms of binding. Said differently, the implementation should be a simple library. Then, it should be usable in various components: in Eclipse (OCCIware Studio), with Erocci (with a D-Bus back-end), within an application server (Apache Tomcat), etc.

It means the code in charge of managing monitoring configurations will be usable by several tools. The REST API will be compliant with OCCI Monitoring (and therefore with OCCI Core). If we complete the previous diagram with this information, we get the architecture illustrated...
in Figure 4.8.

Concepts and Consequences

Let’s identify the key features our solution should have.

Managing a configuration is about displaying and editing parameters. So, there should be web forms to manage these parameters. Parameters may designate sensor configurations, or collector configurations (concepts used in OCCI Monitoring are good and easy to understand). So, web forms (1).

Then, managing also means storing configurations (2).

There is also synchronization (3).

Since this tool is a proxy to configure other tools, we cannot exclude manual modifications on these tools (example: directly edit Nagios files). Therefore, the tool should remember the last pushed configurations and be able to detect if another configuration is applied.

Eventually, there is ordering configurations (4). There can be many ways to classify information: project, application, PaaS, IaaS, production, etc. Since such a terminology depends on team habits, the most simple option is to associate configurations with tags. A configuration should have a content (sensors, collectors... as specified by OCCI Monitoring), but also tags, a creation date and history of the modifications.

With this, we see there is much more than just OCCI Monitoring in this tool.

Applying Configurations

Defining sensors and collectors is not enough. These parameters need to be propagated to target tools. Several ways have to be supported:

- Local files: write parameters to local files.
- a specific case of files is generating code, ex. Riemann publisher Closure code or at SaaS level Dropwizard (formerly Yammer) Metrics Java setup
- REST invocation: user the parameters for REST queries.
- SSH: write parameters to remote files through SSH.
Obviously, other ways may appear. How this will be managed is not clear at the moment. As an example, will we have a SSH utility? Use system commands?

Extensibility

Extensibility is about 3 elements:

1. Support new sensors (i.e. allow to configure a new kind of probe).
2. Support new collectors (i.e. allow to configure a new kind of storing collected information).
3. Support new storage (i.e. allow to save configurations with a new solution: database, file, git, etc).

Whether extensions may be deployed at runtime is not determined yet. Since this tool is only about managing configurations (it does not monitor anything), restarting it is not mandatory an issue.

In any case, developing a new extension should not be complicated. Extensions should be registered publicly to favour reuse by a community of users.

Technical Choices & Development Leads

The project should be developed as a new Git repository under the occiware organization on Github. It should be a stand-alone open source project, although it has links with other developments in OCCIware.

Which development language and frameworks? Which project name? Short-term goals and questions.

- Model an extension for OCCI Monitoring with OCCIware’s studio? What is the interest if there is no code generation for the REST implementation.
- Implement collector mixins to push monitoring data in Elasticsearch. This will allow to use tools like Kibana and Graphana to visualize monitoring information. This will serve as an example to write other collector mixins.
- Implement sensor mixins to configure Nagios / SNMP agents. This will serve as an example to write other sensor mixins. We might also write one for JMX.
- Maybe implement a web interface to configure monitoring. It means hiding REST invocations of OCCI Monitoring.
- Maybe implement a file-checker that would push monitoring configuration from files (most likely the best approach for system administrators).
- Last, but not least, write an implementation of OCCI Monitoring. This implementation should be agnostic in the sense of its integration. It means it should be available within the studio, in Erocci and in other shapes as well. Having it available as a Java library (that can be bound to various integrations stubs) may be a solution.
4.2.4 OCCIware monitoring platform: objectives and architecture

Any IT system with SLA engagements should be monitored. Monitoring keeps track of platforms’ stats and measures, through well-defined indicators, the level of the quality of service (QoS). The OCCIware platform as any platform held by SLA engagements in terms of service continuity and QoS. With this framework we deployed a monitoring platform to oversee the OCCIware platform itself and hosted services.

In the OCCIware project we opted for a generic and flexible solution that allows standard SNMP-based monitoring as well as complex application oriented monitoring: Centreon Enterprise Server, an open source monitoring solution. It measures the availability and performance of the application layer, user services and hardware resources. This monitoring solution brings many features such as service and host status viewing, metrology, reporting, events, advanced user access control (ACL), etc. Additional modules are developed by the editor and the community to extend these functions, e.g., many APIs to interact with Centreon such as the automation of the configuration management. The automation API (CLAPI) does not follow the OCCI recommendations. The challenge would be to develop the OCCI interface that will communicate with the existing API.

Several architectures and configurations are possible for the Centreon monitoring platform. Table 4.5 provides the prerequisites for Centreon architecture depending on the number of monitored services.

<table>
<thead>
<tr>
<th># services</th>
<th>Architecture</th>
<th>Central Server</th>
<th>Poller Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 500</td>
<td>1 central</td>
<td>1 vCPU / 1GB</td>
<td>-</td>
</tr>
<tr>
<td>500 - 2 000</td>
<td>1 central</td>
<td>2 vCPU / 2GB</td>
<td>-</td>
</tr>
<tr>
<td>2 000 - 10 000</td>
<td>1 central + 1 poller</td>
<td>4 vCPU / 4GB</td>
<td>1 vCPU / 2GB</td>
</tr>
<tr>
<td>10 000 - 20 000</td>
<td>1 central + 1 poller</td>
<td>4 vCPU / 8GB</td>
<td>2 vCPU / 2GB</td>
</tr>
<tr>
<td>20 000 - 50 000</td>
<td>1 central + 2 pollers</td>
<td>4 vCPU / 8GB</td>
<td>4 vCPU / 2GB</td>
</tr>
<tr>
<td>50 000 - 100 000</td>
<td>1 central + 3 pollers</td>
<td>4 vCPU / 8GB</td>
<td>4 vCPU / 2GB</td>
</tr>
</tbody>
</table>

Table 4.5: OCCIware IaaS Monitoring Architecture Configurations

In our case we deployed a centralized platform with just one central server:

- Centreon web interface
- Database (MySQL + RRD)
- Monitoring engine
- Broker

This architecture could be easily extended to a distributed one. Several entities are used to implement this architecture:

- The Apache server is responsible for hosting the Centreon web interface
- Multiple MySQL databases that store the Centreon configuration, monitoring information and performance data
- The monitoring engine to monitor IT systems
- The monitoring information are sent through cbmod to Centreon Brocker SQL
• *Centreon Broker SQL* is responsible for inserting the monitoring data in the database and transmit performance data to *RRD Centreon Broker*

• *Centreon Broker RRD* is responsible for generating the *RRD* files (used to generate performance graphs)

Figure 4.9 summarizes these points.

![Monitoring Reference Architecture](image)

**Figure 4.9: Monitoring Reference Architecture**

For scalability reasons, other architectures can be implemented:

• Distributed architecture

• Distributed architecture and remote database

• Redundant distributed architecture

• Redundant distributed architecture + Redundant web interface

**OCCIWare Web Interface Monitoring with Centreon**

Once the OCCIware monitoring platform is deployed we configured the DNS and reverse proxy. The web interface is available on the following URL: [https://monitoring-occiware.cloudsystem.fr](https://monitoring-occiware.cloudsystem.fr) (see Figure 4.10).

**User:** guest

**Password:** guestocciware

The *Centreon* web interface is composed of several menus (not all visible to the guest user), each menu has a specific function (see Figure 4.11):

• The **Home** menu provides access to the first Home screen after logging in. It summarizes the general supervision of the state.

• The **Supervision** menu contains the state of all components monitored in real time and the delayed one through viewing logs.
Figure 4.10: Centreon Web Interface Login Page

- The **Views** menu allows to view and configure the performance graphs for each component of the information system.

- The **Report** menu displays intuitively (charts) monitoring evolutions over a given period.

- The **Setup** menu allows to configure all monitored elements (hosts).

- The **Administration** menu lets the user configure the Centreon web interface as well as view the overall status of servers.

Figure 4.11: Centreon Web Interface Menus

An example of monitoring is already in place; OCCIware monitoring platform supervises itself as shown on Figure 4.12.

In order to facilitate configuration and industrialization, for administrators, an API was developed to enable configurations and command-line deployments: **Centreon CLAPI**.

**Centreon CLAPI and the evolution to OCCI Monitoring**

Centreon CLAPI is an open source module that allows users to deploy the monitoring of their systems in command lines. Centreon CLAPI offers almost all the features. The main features are:

- add/remove/update objects such as “Hosts”, “Services”, “Host Templates”, “Host Groups”, “Contacts”, etc.

- generate configuration files;

- test configuration files;

- push configuration files to the Pollers (satellite servers);
Figure 4.12: Centreon Web Interface Example
• restart pollers;
• import and export objects.

Through CLAPI the automation of the monitoring has become possible without using the web interface. However, the challenge remains to develop an OCCI interface to communicate with CLAPI and with the Centreon Engine to retrieve monitoring information (e.g. CPU rate on a physical server, RAM consumed on a Virtual Machine, the presence of processes in memory, monitoring via specific scripts, service availability, etc.).

4.3 Distributed Deployment

4.3.1 Architecture Schema

The OCCLware runtime (either MART server or erocci with erocci-dbus-java) understands the OCCI requests thanks to his models then those requests are executed by Ozwillo, Roboconf, Proactive Cloud Automation and Scalair which are connected to the OCCLware runtime as backend. The architecture is illustrated by Figure 4.13.

![Runtime Architecture](image)

Figure 4.13: Runtime Architecture

4.3.2 Sequence diagram

Figure 4.14 displays a sequence diagram for a complete chain of the software use.
<table>
<thead>
<tr>
<th>Deliverable</th>
<th>D2.1.2</th>
<th>Status</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected</td>
<td>2016/03/31</td>
<td>Dissemination</td>
<td>Public</td>
</tr>
<tr>
<td>Delivered</td>
<td>December 15, 2016</td>
<td>Version</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Figure 4.14: Sequence Diagram for Distributed Deployment
4.3.3 Ozwillo

Ozwillo is a platform that distributes online services from a variety of SaaS editors. It’s underlying and strategic goal is to forge open, linked and reusable data thanks to the daily use of services.

**Input and Output**

Ozwillo interprets OCCI Linked Data as a Service requests. Then it deploys a LinkedData Node on the infrastructure queried. If there is no infrastructure, Ozwillo asks to the OCCIware runtime an infrastructure to deploy the Linked Data Node.

4.3.4 Roboconf

Roboconf is a platform to manage elastic deployments in the cloud. It manages deployments, probes, automatic reactions and reconfigurations. It can be defined as « PaaS framework »: a solution to build PaaS (Platform as a Service). Most PaaS, such as Cloud Foundry or Openshift, target developers and support application patterns. However, some applications require more flexible architectures or design. Roboconf addresses such cases. With Roboconf, you can create PaaS for any programming language, any kind of applications and any operating system. You define what you put in your platform, you specify all the interactions, administration procedures and so on. Roboconf handles application life cycle: hot reconfiguration (e.g. for elasticity issues) and consistency (e.g., maintaining a consistent state when a component starts or stops, even accidentally). This relies on a messaging queue (currently Rabbit MQ). Application parts know what they expose to and what they depend on from other parts. The global idea is to apply to applications the concepts used in component technologies like OSGi. Roboconf achieves this in a non-intrusive way, so that it can work with legacy software. Application parts use the message queue to communicate and take the appropriate actions depending on what is deployed or started. These appropriate actions are executed by plug-ins (such as bash or Puppet). Roboconf is a distributed technology, based on AMQP and REST/JSON. It is IaaS-agnostic, and supports many well-known IaaS (including OpenStack, Amazon Web Services, Microsoft Azure, VMWare, as well as a “local” deployment plug-in for on-premise hosts).

**Input and Output**

The OCCIware runtime (either MART server or erocci with erocci-dbus-java)’s proxyfication for Roboconf is an OCCI-compliant equivalent of its REST API (OCCI services provider). Roboconf receives REST instructions to manage the life cycle of software components (including infrastructure elements). When infrastructure is required by Roboconf, and that it is available through an OCCI API (served by e.g., MART Server or Erocci), then Roboconf uses its OCCI extension (OCCI services consumer).

**Architecture**

Roboconf relies on 3 elements: the DM, agents and a messaging server (RabbitMQ). VM designate Virtual Machines or any kind of machine (server, device...). Figure 4.15 displays the roboconf architecture.

- The Deployment Manager (or DM) is in charge of managing Virtual Machines (or VM) and the agents. It acts as an interface to the set of VMs or devices. It is also in charge of instantiating VMs.
• The **Agent** is a software component that must be deployed on every VM and device on which Roboconf must deploy or control something. Agents use plug-ins to delegate the manipulation of software instances. As an example, there are a Bash and a Puppet plug-ins. Roboconf does not reinvent the wheel, it tries to reuse existing and robust solutions. Roboconf agents communicate with each other through the messaging server.

• The **Messaging Server** is the key component that enables communications between the DM and the agents. The DM and the agents always communicate asynchronously through this server. Depending on the Roboconf messaging implementation, this server may be available in the DM itself.

Multi-target and hybrid deployments are supported. This can be used for critical periods (load increase) or for migration purpose. It can also serve strategic objectives (like data security and privacy issues). Figure 4.16 illustrates this with some cloud infrastructures. It only shows interactions between Roboconf parts, and not between the parts of the applications that Roboconf deploys.

The overall idea is that the agents run remotely (e.g., in the cloud), while the DM can run in a secured area. The DM is indeed an administration tool that requires a safer installation.

### 4.3.5 Proactive Cloud Automation

The aim of the software is to ease the use of different infrastructures and the migration. As a consequence the deployment can be done through the cloud standard OCCI. The OCCI request will be transformed into a query by the Cloud Automation Server. This query will enable to get a workflow which will be deployed by the scheduler. The workflow contains all the information set by the OCCI request. It will execute this request on the scheduler. The scheduler is connected to several IaaS connectors. In order to manage several providers there are different workflows for each provider. Indeed the IaaS connectors don’t need the same information to deploy an infrastructure.
4.3.6 Input and Output

Proactive Cloud Automation interprets IaaS requests. According to the requests, it will create an infrastructure on the most well suited provider.

Architecture

The Proactive Cloud Automation architecture is depicted into Figure 4.17.

Cloud Automation Server  The Cloud Automation Server is connected with the OCCI infrastructure model and can understand OCCI requests as input. Then it will transform each request into a Workflow Catalog query in order to get the workflow which will deploy an infrastructure complying with the OCCI request. Then this workflow will be deployed on the scheduler. The Cloud Automation Server has a REST API as a consequence it will return, from the requests, an HTTP response that will match with the request execution.

Scheduler  The Scheduler will execute different tasks of the workflow. It executes this computation on a cluster of computation resources, each step on the best-fit resource and in parallel wherever it’s possible. The scheduler uses workflow as an entry then other services can get the result from the workflow computation by querying it.

Workflow Catalog  The catalog is a database which can be access thanks to REST requests, it contains workflows. Basic deployment and lifecycle instances workflows have been developed. Those workflows are dynamically discovered and can be executed. The workflow catalog has a
Figure 4.17: PCA Architecture
REST API which support a SQL language in order to do queries on workflows. The service will return one or several workflows which match with the request.

**Smart Watch**  Smart Watch enables to monitor the infrastructure by gathering the information about their state and their instances state. The service will monitor each endpoint that will be provided as an entry. Then it will raise alert from the endpoint status.

**Multi-IaaS connector**  The multi-IaaS connector enables to do CRUD operations on different infrastructures on public or private Cloud (AWS EC2, OpenStack, VMWare, Docker, etc). It is connected with those infrastructure interfaces in order to manage their lifecycle on virtual machines.

### 4.3.7 Scalair

Scalair is an IaaS provider and offers resources supervision thanks to the dashboard.

#### Input and Output

Scalair receives IaaS requests then it creates the resources on its own infrastructure. It also supplies a web address where resources can be supervised thanks to its dashboard.

### 4.4 OCCI Generic Console - Open API-based frontend

The OCCI Generic Console aims at providing users with intuitive user interface for manipulating OCCI Entities and Categories. While the domain of REST APIs is fastly evolving, standard API documentation formats are emerging, providing out-of-the-box basic API manipulation interfaces.

Amongst them, the OpenAPI initiative[^4] from Linux Foundation has been widely adopted as the *de facto* standard for REST API description. It is based on the Swagger API documentation format[^5].

It has been then decided to include a Swagger/OpenAPI description of the OCCI API in the OCCI runtime (erocci) so that we simply use Swagger frontend for basic manipulation of OCCI Entities and Categories.

#### 4.4.1 OpenAPI OCCI Implementation

The erocci runtime provides an OpenAPI description of the Query interface ("/-/") and of categories collection. Due to the dynamic aspect of the OCCI API, the Swagger description is automatically generated by erocci runtime with regard to supported categories.

An example of such generated Swagger description is available in Appendix A.

#### 4.4.2 OpenAPI frontend

Thanks to the use of standard Swagger-based API description, any compatible frontend can be used as a generic console for OCCI runtime.

The default JavaScript Swagger frontend is provided with erocci runtime. It is available at the /api-docs/ location of the runtime.

[^4]: OpenAPI Initiative website: [https://openapis.org/](https://openapis.org/)
[^5]: Swagger website: [http://swagger.io/](http://swagger.io/)
This default frontend is illustrated on Figure 4.18. Another example of Swagger frontend is illustrated at Figure 4.19 (available at https://github.com/jensoleg/swagger-ui).

Figure 4.18: Default Swagger Frontend

4.5 OCCI Generic Console - Playground

4.5.1 Goal

The Open API-based frontend is generic allright and useful to developers, since it allows them to tailor and test OCCI HTTP requests including headers for their needs. However, it does not target more functional uses. More specifically, it does not allow to easily discover what OCCI resources are available in the server, nor browse and navigate through them, which is a prerequisite before being able to use them. This is all the more a pity because several OCCI features (such as links) would be very useful for that.

That’s why the OCCInterface OCCI functional Playground aims at providing OCCI API users such a simple, intuitive, generic, no-frills (because stays close to JSON) but powerful online tool that allows them to browse OCCI resources in the way REST APIs - and OCCI in particular - are meant to, and to learn about them through examples, manage them and even execute their actions. It is at once:

1. A "Getting started" with a set of tutorials that are one click away from being executed.
2. The "API equivalent of the shell" or SQL REPL command-line interface for Unix or database developers.
3. A "reference documentation" where to document all the capabilities of a cloud implementation or of an integration of clouds.

4.5.2 Deployment use cases

OCCIInterface strives at working with any OCCI implementation, even beyond OCCIware’s ones, but aims at being able to provide additional, optional functionalities in the case of it being an OCCIware runtime.

Another important point is that it supports two modes of deployment:

1. It can be deployed **standalone** thanks to a simple node.js server. In this case, it can be deployed by cloud consumers and enriched with examples of their use cases and scenarii within their integration of clouds.

2. The OCCI Playground is architected to be able to be **embedded** within and served by any OCCI implementation thanks to being developed as browser-side only HTML/JavaScript. In this case, it can be enriched by developers of cloud implementation or extensions with examples of their cloud implementation or extension features.
4.5.3 Features

The OCCI Playground aims at providing the following functionalities:

Allow to browse to resources by listing OCCI API root paths:
• i.e. extension schemes, and below kinds and mixins
• but also user-defined mixins ("tags", the only mixins allowed to be defined in Configuration)

Allow to browse to resources starting from other resources, by "toolifying" JSON results of previous calls to the OCCI API,
• i.e. making HTML links out of HTTP URIs embedded in results (except for scheme URLs)
• up to generating OCCI queries on values embedded in results, that go back to resource collections or type definition

Allow to edit and execute:
• in an URL bar, HTTP queries generated by all those previous functionalities
• results returned by those, and POST, PUT or DELETE them
• and also execute OCCI actions

Allow to set transversal / non-functional headers, notably for:
• representation / rendering (Content-Type and Accept HTTP headers)
• authentication (Authorization or custom HTTP headers)

As much as possible, generate OCCI queries with possible query filters (by kind, by other attributes...) and pagination. However this largely depends on capabilities of OCCI runtimes and backends.

4.5.4 Illustration

The OCCInterface OCCI functional Playground is available at https://github.com/occiware/OCCInterface, publicly deployed and testable at http://occinterface.herokuapp.com and illustrated by Figure 4.20.

4.6 IaaS Administration Console

To meet the standards in terms of ergonomy and reusability, the OCCI IaaS Administration Console uses mainly JavaScript, Bootstrap, Jquery, Nodejs, less and ejs technologies.

In order to easily manage OCCI concepts by developers, we decided to develop a dedicated console, called IaaS Administration Console or Infrastructure Console. This console is used to manage infrastructure resources, which OCCI concepts are more concrete.

This infrastructure console is illustrated in Figure 4.21. A detailed description can be found on Scalair’s project tracking tool.[6]

To use the IaaS console, users must log in. Rights are associated with the user account to allow or not certain actions.

IaaS console’s main objective is to provide a graphical representation of the various resources managed by the OCCIware platform (e.g. applications, infrastructure resources, monitoring, event correlation, etc.).

4.6.1 Main IaaS Administration Console Components

Menu

The menu allows users to select widgets, to choose the application skin by using themes. A link is given to install the Android app. Under this menu we found also the version of the application. The dashboard menu allows adding and deleting (clicking on the trash icon) widgets from the list. Some widgets can be resized using lower right corner as illustrated in Figure 4.23.

Widgets

**Alarm Widget** Alarms from the infrastructure are displayed line by line. Icons, colors and text information depend on the content of the alarm. Generally, we have the host ID sending the alarm, alarm’s duration, output sent by the sensor that triggered the alarm and network (IP) address of the host.

The default settings are:

- The severity of the alarm is indicated by a colored spot;
- The entire alarm text is displayed;
- A counter indicating the total number of alarms is displayed This widget can be configured by clicking on the gear icon on the title bar.

**Virtual Machine/Physical Server Widget** The title of the widget includes, from left to right as shown in Figure 4.23.
1. Shortcuts to help, logout, account preferences and to report bugs or requests by mail.

2. Widgets tab for the console, Administration for user’s account management and Traces for log messages.

3. From top to bottom the action icons to: hide the banner, update all widgets and access the menu.

4. Space dedicated to widgets.

Figure 4.21: Scalair’s IaaS Administration Console
Figure 4.22: IaaS Administration Console Menu

Figure 4.23: Remove or Resize a Widget
• The alarm status for this server is represented by a colored square corresponding to the severity and it contains the total number of alarms related to the server;

• Access to the resource graphs page;

• Access to the monitors page Access to the vulnerability analyzer.

The default view contains real time properties of the Virtual Machine/Physical server:

• Host ID

• Host state: on, off or suspended

• Host actions: start/stop, suspend and restart
- Real-time information on CPU, RAM and Disks (partitions)

This page is updated every minute.

Figure 4.26 shows a real-time monitoring as given by the monitoring system (Centreon/Nagios). The icon and color of the monitor name indicate its severity.

![Figure 4.26: Associated Monitors with a Real Time Output](image)

The vulnerability scanner allows the user to view the latest reports and launch a new analysis as we can see it on Figure 4.27.

![Figure 4.27: Vulnerability Scanner Module](image)

On each virtual machine/physical server resources (i.e., cpu, ram and disk) graphs are associated as we can see it on Figure 4.28. Graphs are on a predefined period: 7 hours, 1 day and one week. Resources and widget are checked and updated every 5 minutes.

**Monitoring Widget**  This widget displays the entire virtual machine/physical server park (i.e., one square = vm/server). It represents both the state of hosts (on, off and suspended) and associated alarms, i.e., the highest severity alarm by host. On Figure 4.29, we can see three hosts (e.g., vm) with warning alarms and one host with a critical alarm.

Other views of the same widget were developed to better manage the park as shown in Figure 4.30.
Figure 4.28: CPU, RAM and Disk(s) Graphs

Figure 4.29: Monitoring Widget
Figure 4.30: Other Monitoring Views
Chapter 5

Use Cases

5.1 DCaaS Use Case Description

Any application requires IT resources to perform and meet different demands assigned to it. Computer resources are materialized by processing units, memory (RAM), storage, networks, protections (firewall), etc. These resources are typically grouped in servers. A server may be a personal computer as it can be a set of special servers housed in one or several data centers – Cloud or MultiCloud. A server may be also a Virtual Machine (VM) on a personal computer or multiple VMs (or containers) hosted on one or several physical servers, therefore in one or more data centers.

Hosted applications are inherently applications that require flexibility and rapid deployment of resources depending on the criticality of resources and the use of the application. The objective is to deliver an elastic cloud infrastructure with fast deployment resources to support applications that might be demanding in terms of resources under certain time constraints.

A cloud-based application (web) could receive variable amounts of requests that vary over time. In this use case we will work on a carpool application. This type of application could be highly stressed during strike periods or on weekends. So an appropriate and scalable infrastructure is very important.

Figure 5.1 shows the infrastructure architecture and the network flow for the carpool application we will implement. The access to the application is done through a single public IP address then the virtual IP 172.16.225.50. This virtual IP is used by the HAproxy cluster; responsible for load balancing (and other features) between the different web servers (172.16.225.71, 172.16.225.72 and 172.16.225.73); the web servers share the same NFS volume for users’s upload files and use different types of data bases: Neo4J, MySQL and MongoDB. As we can see in Figure 5.1, everything is in a cluster configuration (2 Haproxy, 3 web servers, 3 data nodes for Neo4J, 2 data nodes for MySQL, 2 data nodes for MongoDB) to ensure high availability.

This configuration is a basic one; if the load increases the idea would be to increase the number of servers (horizontal scalability) or to increase resources by server (vertical scalability).

5.1.1 Elasticity System and Automatic Deployment

In a typical cloud infrastructure, resources are provisioned in advance and often manually. During a load increase, resources are adjusted to meet the needs but without any anticipation and often at the explicit request of the client.

In this use case we propose a solution to make the cloud elastic; a cloud that adapts to the application and not the reverse.
Figure 5.1: Infrastructure Architecture for Carpool Application
Figure 5.2: Elastic Carpool Application with ProActive and Roboconf
In Figure 5.2 we resume our architecture of the carpool application, but this time with new elements to ensure application scalability:

- **Elasticity Manager** will be able to monitor the inflows and the behavior of the different servers. The supervision of these elements will enable platform to anticipate requests’ overload and make decisions to increase or to add additional resources. The Elasticity Manager makes decisions based on the Monitoring Manager outputs.

- **Roboconf** will be able to provide hot (re)configuration of application nodes. For example, when a new node appears or disappears due to elasticity decisions (e.g., load-balancing, etc.), the configuration of dependent application nodes will be updated (e.g., when adding a new application server node, it will be automatically configured to connect to already existing databases, and load balancers will be reconfigured to redirect traffic to the new node). The process is asynchronous, and depends only on the application’s topology: for example, when autonomic decisions are made by Proactive’s cloud manager (it decides to add or remove a node), Roboconf will perform the adequate application reconfiguration, so the global application still works.

- **ProActive Cloud Automation** will be able to make decisions concerning elasticity (e.g., add or remove nodes), based on criteria like application load, latency, SLA, etc., using a decision-making algorithm. Then, it will be able to make these decisions operational, and change the application’s topology to adapt to the new situation.
Chapter 6

Conclusion

While the Open Cloud Computing Interface specification describes standard models and renderings for manipulating cloud computing resources, the reuse of Open Cloud Computing Interface-compliant tools is really limited by the absence of formal, designing and runtime tools.

In this document, we have described the architecture of a comprehensive toolchain made of several components with clear interfaces and role, so that every developer involved in the development of an Open Cloud Computing Interface-compliant application can use some or all of them, together or independently.

While the objectives of OCCIware project require a multi-disciplinary approach, they also bring the risk of building inconsistent toolchain, with limited interactions between components. The designed architecture is dedicated to mitigating this risk.

The usability of the toolchain will be validated by the use cases and their feedback will be used to adjust the architecture wherever relevant.
<table>
<thead>
<tr>
<th>Deliverable</th>
<th>D2.1.2</th>
<th>Status</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected</td>
<td>2016/03/31</td>
<td>Dissemination</td>
<td>Public</td>
</tr>
<tr>
<td>Delivered</td>
<td>December 15, 2016</td>
<td>Version</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Appendix A

OCCI Swagger Description

```json
{
  "swagger": "2.0",
  "paths": {
    "/categories/storagelink": {
      "post": {
        "tags": ["http://schemas.ogf.org/occi/infrastructure#storagelink"],
        "responses": {},
        "produces": ["application/xml",
                      "application/occi+xml",
                      "application/json",
                      "application/occi+json",
                      "text/occi",
                      "text/plain"],
        "parameters": [],
        "description": "Creates a new entity the kind http://schemas.ogf.org/occi/infrastructure#storagelink (StorageLink)",
        "consumes": ["application/xml",
                      "application/occi+xml",
                      "application/json",
                      "application/occi+json",
                      "text/occi",
                      "text/plain"]
      },
      "get": {
        "tags": ["http://schemas.ogf.org/occi/infrastructure#storagelink"],
        "responses": {},
        "produces": ["application/xml",
                      "application/occi+xml",
                      "application/json",
                      "application/occi+json",
                      "text/occi",
                      "text/plain",
                      "text/uri-list"],
        "parameters": [],
        "description": "Retrieve the collection of entities of the kind http://schemas.ogf.org/occi/infrastructure#storagelink (StorageLink)",
        "consumes": []
      },
      "delete": {
        "tags": ["http://schemas.ogf.org/occi/infrastructure#storagelink"],
        "responses": {},
        "produces": [],
        "parameters": [],
        "description": "Remove entities of the kind http://schemas.ogf.org/occi/infrastructure#storagelink (StorageLink)",
        "consumes": []
      }
    },
    "/−/":
  }
}
```

"get":
{
  "tags": ["Query Interface"],
  "responses": {},
  "produces": ["application/xml",
               "application/ooci+xml",
               "application/json",
               "application/ooci+json",
               "text/ooci",
               "text/plain",
               "text/uri-list"],
  "parameters": [],
  "description": "Retrieve Category instances",
  "consumes": []
},
"delete":
{
  "tags": ["Query Interface"],
  "responses": {},
  "produces": [],
  "parameters": [],
  "description": "Remove a user-defined Mixin instance",
  "consumes": ["application/xml",
               "application/ooci+xml",
               "application/json",
               "application/ooci+json",
               "text/ooci",
               "text/plain"]
}
}

"info":
{
  "title": "API-DOCS",
  "basePath": "",
}
Bibliography