<table>
<thead>
<tr>
<th><strong>GRANT AGREEMENT N.</strong></th>
<th>688386</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PROJECT REF. NO</strong></td>
<td>H2020- 688386</td>
</tr>
<tr>
<td><strong>PROJECT ACRONYM</strong></td>
<td>OPERA</td>
</tr>
<tr>
<td><strong>PROJECT FULL NAME</strong></td>
<td>LOw Power Heterogeneous Architecture for Next Generation of SmaRt Infrastructure and Platform in Industrial and Societal Applications</td>
</tr>
<tr>
<td><strong>STARTING DATE (DUR.)</strong></td>
<td>01/12/2015</td>
</tr>
<tr>
<td><strong>ENDING DATE</strong></td>
<td>30/11/2018</td>
</tr>
<tr>
<td><strong>PROJECT WEBSITE</strong></td>
<td><a href="http://www.operaproject.eu">www.operaproject.eu</a></td>
</tr>
<tr>
<td>**WORKPACKAGE N.</td>
<td>TITLE**</td>
</tr>
<tr>
<td><strong>WORKPACKAGE LEADER</strong></td>
<td>IBM</td>
</tr>
<tr>
<td>**DELIVERABLE N.</td>
<td>TITLE**</td>
</tr>
<tr>
<td><strong>RESPONSIBLE AUTHOR</strong></td>
<td>Dr. Alberto Scionti</td>
</tr>
<tr>
<td><strong>DATE OF DELIVERY (CONTRACTUAL)</strong></td>
<td>31/05/2017 (M18)</td>
</tr>
<tr>
<td><strong>DATE OF DELIVERY (SUBMITTED)</strong></td>
<td>31/05/2017</td>
</tr>
<tr>
<td>**VERSION</td>
<td>STATUS**</td>
</tr>
<tr>
<td><strong>NATURE</strong></td>
<td>R(Report)</td>
</tr>
<tr>
<td><strong>DISSEMINATION LEVEL</strong></td>
<td>PU(Public)</td>
</tr>
<tr>
<td><strong>AUTHORS (PARTNER)</strong></td>
<td>A. Scionti (ISMB), P. Ruiu (ISMB), J. Nider (IBM), M. Rapoport (IBM), L. Scanavino (CSI), G. Urlini (ST)</td>
</tr>
<tr>
<td>VERSION</td>
<td>MODIFICATION(S)</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------</td>
</tr>
<tr>
<td>0.1</td>
<td>First Draft</td>
</tr>
<tr>
<td>0.2</td>
<td>Revised structure and filled contents</td>
</tr>
<tr>
<td>0.3</td>
<td>Reviewed by consortium’s partners (CSI, ST)</td>
</tr>
<tr>
<td>1.0</td>
<td>Final version</td>
</tr>
<tr>
<td>PARTICIPANTS</td>
<td>CONTACT</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| STMICROELECTRONICS SRL | Giulio Urlini  
Email: Giulio.urlini@st.com |
| IBM ISRAEL SCIENCE AND TECHNOLOGY LTD | Joel Nider  
Email: joeln@il.ibm.com |
| HEWLETT PACKARD CENTRE DE COMPETENCES (FRANCE) | Gallig Renaud  
Email: gallig.renaud@hp.com |
| NALLATECH LTD | Craig Petrie  
Email: c.petrie@nallatech.com |
| ISTITUTO SUPERIORE MARIO BOELLA | Olivier Terzo  
Email: terzo@ismb.it |
| TECHNION ISRAEL INSTITUTE OF TECHNOLOGY | Dan Tsafrir  
Email: dan@cs.technion.ac.il |
| CSI PIEMONTE | Vittorio Vallero  
Email: Vittorio.vallero@csi.it |
| NEAVIA TECHNOLOGIES | Stéphane Gervais  
Email: s.gervais@lacroix.fr |
| CERTIOS BV | Frank Verhagen  
Email: frank.verhagen@certios.nl |
| TESEO SPA | Stefano Serra  
Email: s.serra@teseo.clemessy.com |
| DEPARTEMENT DE L’ISERE | Olivier Latouille  
Email: olivier.latouille@isere.fr |
## ACRONYMS LIST

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSS</td>
<td>Open Source Software</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interfaces</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>CLI</td>
<td>Command Line Interface</td>
</tr>
<tr>
<td>YAML</td>
<td>Yet Another Markup Language</td>
</tr>
<tr>
<td>HOT</td>
<td>Heat Orchestration Template</td>
</tr>
<tr>
<td>ECRAE</td>
<td>Efficient Cloud Allocation Engine</td>
</tr>
<tr>
<td>AMQP</td>
<td>Advanced Message Queuing Protocol</td>
</tr>
<tr>
<td>CRIU</td>
<td>Checkpoint/Restore In Userspace</td>
</tr>
</tbody>
</table>

*Table 1 List of Acronyms*
LIST OF FIGURES

Figure 1 - OpenStack general overview of the control plane and APIs. .......................................................... 11
Figure 2 - OpenStack communication flow between Compute and Orchestration services. .......................... 12
Figure 3 - High-level overview of the OpenStack core services and their relationship with each other. ... 12
Figure 4 - General architecture and components of the Compute Service. .................................................. 13
Figure 5 - Structure of the tables used to associate respectively tags to flavours, and flavours to the actual configuration of the nodes (the fields on the Flavour Description table are representative of the information that can be access, but they may vary on the actual implementation). .................................................. 14
Figure 6 - General architecture and components of the Orchestrator Service................................................. 15
Figure 7 - OPERA application descriptors flow. ............................................................................................ 16
Figure 8 - The TOSCA model for the OwnCloud application. ...................................................................... 17
Figure 9 - OPERA scheduling chain for cloud applications. .......................................................................... 17
Figure 10 - Querying Knowledge Base and selecting the executing node. .................................................. 18
Figure 11 - Example of the current Knowledge Base (KB) implementation. ............................................. 19
Figure 12 - Testbed setup with 3 physical nodes configured to run OpenStack components, and hosting VMs and Containers. ........................................................................................................ 21

LIST OF TABLES

Table 1 List of Acronyms .............................................................................................................................. 4
Table 2 List of Actions and Roles .............................................................................................................. 8
ACKNOWLEDGEMENT

This report forms part of the deliverables from a project called “OPERA” which has received funding from the European Union’s H2020 Framework Programme under agreement 688386.

OPERA aims at supporting ambitious challenges on Designing next generation Low Power and Ultra Low Power (ULP) systems, improving energy efficiency in computing by means of heterogeneous architecture, providing a smart and energy efficient solution for the interaction between embedded ULP smart systems and remote small Form Factor Data Centers.
**1 EXECUTIVE SUMMARY**

Work Package 5 (WP5) aims at providing a mechanism to efficiently deploy applications on the data centre resources. The “efficiency” expected from the proposed mechanism strictly concerns the energy consumed by the data centre hardware resources and the relative performance of the various tasks composing applications running on the data center. Providing this mechanism requires to take into account several elements that influence the runtime execution of such tasks. For instance, a model to evaluate the best mapping of tasks with specific host architectures, a mechanism to select the best host among various with the same architecture where to run the tasks, and a way of optimizing the whole data center workload should be integrated.

**1.1 POSITION OF THE DELIVERABLES IN THE WHOLE PROJECT CONTEXT**

The activities carried out within task T5.3, whose initial results are summarized in this document, are framed in the WP5 – Optimized Workload Management on Heterogeneous Architecture. Specifically, this deliverable provides the results of the research activity and investigation done by OPERA partners, aiming at integrating a software system with the OpenStack orchestration toolchain, which is responsible for the mapping of cloud application components on heterogeneous resources. Deliverable D5.4 provides a depth insight of the internal algorithms used by such module. We refer to it as the Efficient Cloud Resources Allocation Engine (ECRAE).

The deliverable is in connection with the work done in the WP5, specifically with activities reported in D5.4 and D5.1. Regarding the connection with D5.4, this deliverable provides a depth insight and analysis of the OpenStack modules’ interfaces that will be leveraged by the ECRAE module to instantiate application components. To this end, the deployment toolchain is analyzed and described. D5.1 provides the analysis of the TOSCA application descriptor format used by ECRAE to allocate each application components on the most suitable hardware, as well as it provides an analysis of how the workload can be characterized.

Finally, the activities belonging to the task T5.3 provides important and useful results that are at the basis of the activities carried out in WP7, specifically related to the VDI use case. Similarly, activities carried out in WP4 are of interest for this task, since the way energy efficiency is evaluated provides the correct input to drive the decision of the developed ECRAE module, as well as it provides input to better tune the entire deployment toolchain.

**1.2 DESCRIPTION OF THE DELIVERABLE**

The task 5.3 aims at investigating and implementing a software module which permits deployment of a distributed application on the most suitable hardware resource. In our vision, a distributed application is a set of software modules which communicate each other through the network (mostly by means of REST API), which is also the general setting of a microservices. Since OpenStack has been chosen as cloud computing manager, one of the aim of this document is describing the modules designed to handle this functionality, such as controlling compute, scheduling resources, deploying instances, migrating workloads and monitoring status of the infrastructure. The practical goal of this task is to extend native functionalities of OpenStack with extra capabilities, tailoring the current modules to OPERA objectives. Specifically, the activities regard integrating, customizing or replacing native software with new code in order to place instances on the most suitable architecture, taking into account consumption profiles of resources. A prototype of a heterogeneous architecture scheduler for OpenStack with focus on energy efficiency will be implemented.

ISMB is the task leader since it owns competences on cloud platform (OpenStack), TECH and IBM participate in the development of specific modules and functions for the interaction with operating system/hypervisor level. CSI provides support for real cloud computing application requirements. The final version of the software D5.3 – Cloud software interface, will be delivered on M34 after the refinement cycles defined in the project.
1.3 LIST OF ACTIONS AND ROLES

Activities related to D5.3 involved mainly TECH, IBM, CSI, and ISMB partners. However, also other partners provided useful inputs for driving the activities carried out in T5.3.

<table>
<thead>
<tr>
<th>ACTIVITIES LIST AND PARTNERS ROLES</th>
<th>CERTIOS</th>
<th>CSI</th>
<th>HPE</th>
<th>IBM</th>
<th>ISMB</th>
<th>LD38</th>
<th>NALLATECH</th>
<th>NEAVIA</th>
<th>ST</th>
<th>TECHNION</th>
<th>TESEO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis of Cloud Application descriptors and their comparison</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOSCA extension and usage</td>
<td>P</td>
<td>I</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenStack analysis and installation on a testbed</td>
<td>P</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analysis of the interfaces of OpenStack modules and design of the deployment toolchain</td>
<td></td>
<td>I</td>
<td>I</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contribution to the deliverable organization, writing, and reviewing</td>
<td>R</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R</td>
<td></td>
</tr>
</tbody>
</table>

Table 2 List of Actions and Roles

a) P = Participating (includes I & R)
b) I = Input delivery (Includes R)
c) R = review

**ISMB:** it is the main contribution for the activities carried out in the task T5.3. ISMB started with the analysis of the different cloud application descriptors, and their comparison. TOSCA has been chosen as the standard format since it provides a vendor-agnostic solution for describing cloud applications in a structured format. CSI also provided inputs useful to better understand how to better exploit the TOSCA format for the OPERA purposes.

**CSI:** mainly contributed by providing inputs regarding the structure and usage of TOSCA (benefits, advantages, disadvantages, limits of the current standard version), and how to possibly extend it to the purposes of OPERA. CSI provided access to the infrastructure, where OpenStack has been installed and it is used to the purpose of integration (see also WP7 – VDI use case).

**IBM:** contributed to this task, by providing information on how to interface with the CRIU-based mechanism used to manage the Linux containers migration. IBM also contributed actively in the organization and writing of this document.

**STM:** reviewed the material provided in this deliverable.
# TABLE OF CONTENTS

1 EXECUTIVE SUMMARY ................................................................. 7  
   1.1 POSITION OF THE DELIVERABLES IN THE WHOLE PROJECT CONTEXT .................................................. 7  
   1.2 DESCRIPTION OF THE DELIVERABLE ........................................................................................................ 7  
   1.3 LIST OF ACTIONS AND ROLES ..................................................................................................................... 8  
2 WHY AN ORCHESTRATOR? ....................................................... 10  
3 OPENSTACK OVERVIEW ......................................................... 11  
   3.1 COMMUNICATION MODEL ............................................................................................................................. 11  
   3.2 NATIVE MODULES .......................................................................................................................................... 12  
   3.3 NOVA COMPUTE .......................................................................................................................................... 13  
   3.4 FLAVORS ...................................................................................................................................................... 14  
   3.5 ORCHESTRATION SERVICE ........................................................................................................................ 14  
4 OPERA TOOLCHAIN ..................................................................... 16  
   4.1 OPERA APPLICATION DESCRIPTOR ............................................................................................................. 16  
   4.2 OPERA APPLICATION DEPLOYMENT SYSTEM ............................................................................................. 17  
      4.2.1 Static Resources Allocation Mechanism ............................................................................................... 18  
      4.2.2 Knowledge Base .................................................................................................................................... 19  
      4.2.3 Dynamic Resources Allocation Mechanism ......................................................................................... 19  
5 TESTBED ENVIRONMENT ......................................................... 21  
6 WORKLOAD MIGRATION ............................................................ 22  
   6.1 COMPARING POST-COPY AND PRE-COPY MIGRATION APPROACHES .................................................. 22  
      6.1.1 Userfaultfd ............................................................................................................................................. 23  
   6.2 CRIU COMMAND LINE INTERFACE ........................................................................................................ 23  
   6.3 CRIU MIGRATION PROCESS ......................................................................................................................... 23  
   6.4 LXC CONTAINER MIGRATION ....................................................................................................................... 24  
7 CONCLUSIONS ............................................................................. 25  
8 REFERENCES ................................................................................ 26
Cloud computing paradigm is based on the availability of a large set of compute, storage and networking resources. These resources, which are in general heterogeneous in nature, need to be controlled in such way user requirements are met and resource usage is maximized. Recently, with the growing demand for more energy-aware systems (reducing energy consumption is essential to sustain the growing demand for computing and storage resources) [1, 2, 3, 4], heterogeneity in data centers has been further increased by introducing different processor architectures and dedicated accelerators. The former are well represented by the growing presence of ARM and POWER8-based systems, while the latter are represented by GPUs and recently the support for FPGA devices [13, 14].

One of the main tasks in governing such kind of infrastructure is the allocation of resources for the different applications, and in order to be energy-efficient, such allocation should consider different additional factors to get the optimal allocation, such as power consumption of each platform and the relative load.

OPERA aims at improving current systems and tools used to orchestrating the applications in the cloud environment. Orchestration generally refers to the automation of the actions required to deploy an application on the cloud infrastructure, in particular it concerns the allocation of resources, the installation of application components on such acquired resources, and the needed configuration of application components and infrastructural elements. In this perspective, the allocation of resources by optimizing the energy efficiency of the cloud infrastructure represents a key element. To this end, we aim specifically to enhance the OpenStack system (see also deliverable D5.4) in such a way that it is possible to:

- Deploy the application components on the most suitable platform (i.e., by choosing the better processor architecture, amount of memory, storage space, etc.);
- Periodically rescheduling (thus migrating) the application components to the most suitable platform if different load/efficiency conditions arise (e.g., a web-frontend previously running on an ARM-based server can be moved to a X86_64 machine if the load of the ARM machine exceeds a threshold and/or the number of requests to the frontend increased).

To this end, we refer to the first point as a **static deployment** action, while we talk of **dynamic scheduling** of the application components in the second bullet point. A software system, referred to as Efficient Cloud Resources Allocation Engine (ECRAE), specifically designed to manage such actions is described in a complementary deliverable (see D5.4). In order to perform these two actions, the cloud system needs to be able to match each application component with the most suitable platform based on the indication collected in a Knowledge Base (KB), and to monitor the status of the infrastructure. Moreover, it is necessary that the application can be split into independent components, as well such that each component has been somehow profiled. To manage the whole orchestration process, we rely on a standardized application description format (TOSCA) which encapsulates all the details needed to correctly assign the components on the specific platform, and to correctly configure them.
3 OPENSTACK OVERVIEW

OpenStack is an open source project which aims at providing a set of tools and software components for managing private cloud infrastructures. In particular, OpenStack provides the set of software components needed to managing a IaaS (Infrastructure-as-a-Service) cloud. The project is declared as open source, open design, open development, and open community. It has been created to be flexible and modular, so that any component can be instantiated or not in the cloud infrastructure irrespective of the others (actually a minimal set of components is required for a minimal installation). Different modules of the suite provide specific services that allow to create a cloud operating system that controls a large pool of compute, storage, and networking resources in a data centers (see Figure 1).

![OpenStack general overview of the control plane and APIs.](image)

OpenStack components are independent and loosely coupled in order to ensure simple extension, interoperability and scalability of the architecture. Some of the design principles which enable these capabilities are:

- Queue based communication
- Asynchronous everything (don't block)
- Independent scalability of components and allow visibility into internal state

OpenStack modules are written in Python, a well-supported and widely adopted programming language, and they provide different APIs to interact with the module (it is worth to note that modules can make use of agents installed on the nodes to perform their job). In general, such APIs are accessed through a client or they can be used in the form of REST-APIs.

Among the various OpenStack modules available, in the context of OPERA we prepared a testbed installation for performing some experiments and supporting the development of new features/extensions.

3.1 COMMUNICATION MODEL

OpenStack consists of several independent parts, named the OpenStack services. All services authenticate through a common identity service. Individual services interact with each other through public APIs, except where privileged administrator commands are necessary.

Internally, OpenStack services are composed of several processes. All services have at least one API process, which listens for API requests, preprocesses them and passes them on to other parts of the service. With the exception of the identity service, the actual work is done by distinct processes.

For communication between the processes of one service, an AMQP message broker is used. The service’s state is stored in a database. When deploying and configuring an OpenStack cloud, you can choose among several message broker and database solutions, such as RabbitMQ, MySQL, MariaDB, and SQLite. Users can access OpenStack via the web-based user interface implemented by Dashboard, via command-line clients and by issuing API requests through tools like browser plug-ins or curl. For applications, several
SDKs are available. Ultimately, all these access methods issue REST API calls to the various OpenStack services. In Figure 2 an extract of the communication model is represented, with focus on the two most relevant services for the OPERA project, namely Compute and Orchestration.

![Figure 2 - OpenStack communication flow between Compute and Orchestration services.](image)

### 3.2 NATIVE MODULES

OpenStack provides an Infrastructure-as-a-Service (IaaS) solution through a set of interrelated services. Each service offers an application programming interface (API) that facilitates this integration. Depending on your needs, you can install some or all services. In Figure 3 an overview of the main components of the platform is shown.

![Figure 3 - High-level overview of the OpenStack core services and their relationship with each other.](image)
For the purpose of OPERA not all the services will be considered, since only the ones involved in the deployment process are relevant for the objectives of the project. In particular the Compute and the Orchestration service will be analyzed in details in the next sections.

The Compute service (Nova), manages the lifecycle of compute instances in an OpenStack environment, spawning, scheduling and decommissioning virtual instances on demand.

The Orchestration service (Heat), orchestrates multiple composite cloud applications by using either the native HOT template format or other template format, through both an OpenStack-native REST API and a external-compatible Query API.

### 3.3 NOVA COMPUTE

OpenStack Compute serves as the core of the OpenStack cloud by providing virtual machines on demand. Compute schedules virtual machines to run on a set of nodes by defining drivers that interact with underlying virtualization mechanisms, and by exposing the functionality to the other OpenStack components. Compute supports the libvirt driver libvirt that uses KVM as the hypervisor. The hypervisor creates virtual machines and enables live migration from node to node.

As per all the other modules of OpenStack, also compute must interact with the identity service to authenticate instance and database access. Furthermore, in order to access images and launch instances, it communicates with the image service.

Compute is a complex software component composed of different modules interacting each other and with external modules, as represented in Figure 4. Each functionality can be controlled from the GUI or using the CLI. The following is a list of the main functionality of this module:

- `openstack-nova-api`, handles requests and provides access to the Compute services, such as booting an instance
- `openstack-nova-cert`, provides the certificate manager
- `openstack-nova-compute`, runs on each node to create and terminate virtual instances. The compute service interacts with the hypervisor to launch new instances, and ensures that the instance state is maintained in the Compute database
- `openstack-nova-conductor`, provides database-access support for compute nodes to reduce security risks
- `openstack-nova-consoleauth`, handles console authentication
● **openstack-nova-network**, network services that can serve as an alternative to OpenStack Networking and handle basic network traffic for private and public access
● **openstack-nova-novncproxy**, provides a VNC proxy for browsers to enable VNC consoles to access virtual machines
● **openstack-nova-scheduler**, dispatches requests for new virtual machines to the correct node based on configured weights and filters
● **nova**, command-line client to access the Compute API

### 3.4 FLAVORS
Flavor is the tool used by OpenStack to specify the size of the instances which have to be deployed. Each created instance is given a flavor, or resource template, which determines the instance size and capacity. Instance flavors allow to measure capacity forecasting, because common use cases are predictably sized and not sized ad-hoc. This specification is relevant for OPERA since workload characteristics can also influence hardware choices and flavor configuration, especially where the tasks present different ratios of CPU, RAM, or HDD requirements. In our vision Flavors permit to couple the application computing demand (i.e. the workload) with the desired capacity (expressed in terms of resources) and the most efficient hardware.

To facilitate packing virtual machines to physical hosts, the default selection of flavors provides a second largest flavor half the size of the largest flavor in every dimension. The flavor has half the vCPUs, half the vRAM, and half the ephemeral disk space. Each subsequent largest flavor is half the size of the previous flavor. The default flavors are recommended for typical configurations of commodity server hardware. To maximize utilization, you might need to customize the flavors or to create new flavors to align instance sizes to available hardware.

As specified in the official documentation it is important to note that Type 1 hypervisors can schedule CPU time more easily to VMs that are configured with one vCPU. For example, a hypervisor that schedules CPU time to a VM that is configured with 4 vCPUs must wait until four physical cores are available, even if the task to perform requires only one vCPU.

<table>
<thead>
<tr>
<th>LOOKUP TABLE</th>
<th>FLAVOUR DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAG</td>
<td>FLAVOUR_1</td>
</tr>
<tr>
<td>tag_1</td>
<td>flv_id_1</td>
</tr>
<tr>
<td>tag_2</td>
<td>flv_id_2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Table**: Structure of the tables used to associate respectively tags to flavours, and flavours to the actual configuration of the nodes (the fields on the Flavour Description table are representative of the information that can be access, but they may vary on the actual implementation).

### 3.5 ORCHESTRATION SERVICE
The OpenStack Orchestration is a service devoted to manage clusters of instances (that can be seen as application), automatically configures environments and deploys resources in stacks. Orchestration stacks are defined with templates, which are non-procedural documents. Templates describe the application in terms of resources, parameters, inputs, constraints, and dependencies in the JSON format. The Orchestration service accept Heat Orchestration Template (HOT) templates that are written in YAML.1

Orchestration can be accessed through a CLI and RESTful queries. The Orchestration service provides both an OpenStack-native REST API and a CloudFormation-compatible Query API. The Orchestration service is also integrated with the OpenStack dashboard to perform stack functions through a web interface.

---

1 YAML is a terse notation that loosely follows structural conventions (colons, returns, indentation), thus is easier to write, parse, grep, generate with tools, and maintain source-code management systems.
The service is composed by different modules and can be controlled from a section of the GUI or by using the CLI (see Figure 6). The following is a list of the main functionality of this module:

- openstack-heat-api, OpenStack-native REST API that processes API requests by sending the requests to the openstack-heat-engine service over RPC
- openstack-heat-api-cfn, optional AWS-Query API compatible with AWS CloudFormation that processes API requests by sending the requests to the openstack-heat-engine service over RPC
- openstack-heat-engine, Orchestrates template launch and generates events for the API consumer
- openstack-heat-cfntools, package of helper scripts such as cfn-hup, which handle updates to metadata and execute custom hooks
- heat, command-line tool that communicates with the Orchestration API to execute AWS CloudFormation APIs

These modules do not need to be modified further to be compatible with OPERA objectives.

![Figure 6 - General architecture and components of the Orchestrator Service.](image-url)
4 OPERA TOOLCHAIN

This chapter describes the main approach developed in OPERA to schedule the execution of the application components on the heterogeneous infrastructure. The proposed approach envisages the introduction of two modules that interface with Heat, Ceilometer and Nova ones to deploy the application components. Specifically, has been described how, starting from a “tagged” version of the TOSCA file, we can generate a complete TOSCA descriptor (i.e., for each application block we determine the flavour where to instantiate that block – e.g., a DBMS), and how we can actually instantiate it on the reserved node.

4.1 OPERA APPLICATION DESCRIPTOR

In order to be independent from a specific platform has been decided to adopt one of the most supported application descriptor, namely the TOSCA descriptor.

TOSCA is a standard created by the OASIS committee to describe application and cloud infrastructural elements in a portable form. The standard provides a way of creating a template that describe all the required elements needed to correctly deploying the application. It originally supports template description as XML files, but a YAML-based version is under validation. TOSCA template can be automatically translated in the native Heat format (i.e., HOT format) through a dedicated translation tool. OpenStack provides such tools as part of the Heat module installation.

TOSCA uses a graph to map application components and their relationships. Nodes allow to hierarchically describe each component up to the lowest level (i.e., the VM or container that host the component), while edges provides the relationships among the components (each edge has a type describing the specific relationship, e.g., hosted on, etc.). Nodes expose capabilities (essentially features that can be used by other nodes to match with their requirements), interface (a set of function that can be used to interact with the node, e.g., a create function that allow to create a database, etc.), and requirements (a set of features that must be satisfied in order to correctly work).

A set of scripts can be used to actually performs specific operation for the nodes. TOSCA has a set of standard node types, but customized nodes can be created. A plan (i.e., a workflow) can be also added to describe the correct order of deploying nodes.

![Figure 7 - OPERA application descriptors flow.](image)

The basic idea is to produce a description of the application, detailed in all distributed modules and relations among them, with an extra information about the specific workload of the single module. This new tag is used by the energy efficiency module to assign a specific hardware profile. In order to be deployed on the platform the descriptor is then reconducted to a standard format of TOSCA inserting the resources selected in the default format. Finally, the descriptor is converted in the format supported by the cloud computing manager (in the case of OpenStack is HOT) and used to deploy the application stack. This flow is represented in Figure 7.

A pilot application has been chosen to validate this solution, specifically OwnCloud\(^2\), one of the most adopted on premises OSS cloud storage application. The aim is to use it as a vehicle to test the technologies, algorithm and tools developed. OwnCloud is composed by two main modules in the basic installation: a web server which provides the front-end for managing the application, and a database to manage data (it is worth to recall that the application provides a storage service for files and documents) stored on the platform. Figure 8 shows the topology of the application, with a generic indication for the hosting nodes for the web front-end and the database.

\(^2\) [https://owncloud.org](https://owncloud.org)
4.2 OPERA APPLICATION DEPLOYMENT SYSTEM

The proposed approach envisages the introduction of two modules that interface with Heat, Ceilometer and Nova ones to deploy the application components. The allocation is based on the evaluation of the system load and other metrics.

Figure 9 shows the main chain used in OPERA to automatically allocate an application on the heterogeneous infrastructure.

The toolchain receives in input the application descriptor, i.e., a file following the TOSCA-Yaml format in which the last level of the hierarchy (i.e., nodes identifying the execution platforms are actually substituted by a general indication of the application profile – a tag – which is used to select the specific architecture) contains the indication of the desired execution platform. The execution platform is actually described by a tag that indicates the profile of the application module. For instance, if the tag indicates a memory intensive application module, the deployment system (i.e., the Energy-aware Deployer) will look for a free compute node equipped with a large amount of memory. This module selects the most appropriate platform based on the application profile indicated by the tag, and by querying the metering module of OpenStack (Ceilometer). This latter can track several parameters (number of instances allocated on the node, volume of RAM allocated and used by a VM instance, number of vCPUs assigned to a VM instance, CPU time used, etc.) from each node of the infrastructure, which can be used to select the correct system. Such measurements can be included in a model for measuring and expressing the energy efficiency of the platform, so that it is possible to rank the platforms based on their effective efficiency. A knowledge base
is also used to initially select the set of possible architecture where to deploy the application module, based on the tag indication. Such knowledge base is organized as database: the tag is the main parameter for querying the DB, and in output the system receives the list of possible architectures to use. Once a platform has been selected, the Energy-aware Deployer generates an output TOSCA-Yaml file in which nodes of the last hierarchical level have been substituted with their actual description (i.e., the architecture, memory, number of CPU to use, etc.). Such model is forwarded to Heat, which in turn interacts with the Nova scheduler to actually deploy the application modules. Again, here a second module can periodically query Ceilometer to assess the status of the infrastructure and eventually plan the migration of application modules from one node to another.

4.2.1 Static Resources Allocation Mechanism

Basically, there are two approaches for using tags:

1. In the first approach, a tag the specifies exactly the architecture to select. This approach is useful for forcing the execution of a specific software module on a specific hardware system. It requires only to extract the information regarding that platform to compose the final TOSCA descriptor file;
2. In the second approach, the tag is used as a hint for selecting the architecture where to run. The tag works thus as a generic indication of the architecture needed to run the software module. This indication is obtained by the programmer during a profiling initial phase. For instance, the tag may indicate a memory intensive software module; the knowledge base should contain for such tag a list of preferred architecture and configuration nodes (e.g., an X86_64 node equipped with large memory). Then, it is responsibility of the Nova scheduler to select one free node with available resources matching with the request.

While the first approach allows to force the execution on a given architecture, the second approach also can leverage on the information collected from Ceilometer. In fact, once a node has been selected, the Energy-aware Deployer can extract the information from Ceilometer and determine the effective energy efficiency of running the module on that node. Similar computation can be applied to the other nodes, so that it is possible to get a ranked list of nodes, from which to select the best element.
Figure 10 shows graphically how the knowledge base is queried for extracting the list of possible destination nodes. In the example, one of the compute node provided with the tagged application descriptor is tagged as “tag_2”. Accessing the KB as a LUT, the Energy-aware Deployer extract a list of possible target node (i.e., from ARCH_1 to ARCH_n). Such list can be ordered depending on the evaluation of each target node against an efficiency model. This model can use input parameters gathered from Ceilometer to infer a numerical indication of the (energy) efficiency of running the software module on that node, given the current working condition of the node. Depending on the results for each target node, the one with the highest efficiency is selected, and the corresponding TOSCA model is integrated in the application descriptor file.

4.2.2 Knowledge Base

The knowledge base is essentially a relational database storing a set of structured information. At its basis, it should work as a look-up table (LUT), where tag elements are used to extract a list of possible architectures and configurations.

In order to correctly select the execution node, a simple ranking model is necessary. This model should use information collected from Ceilometer regarding the current working conditions of the selected node. For instance, if an X86_64 node is selected, the model should be able to provide an indication of how effective (having in mind we need to maximize the energy efficiency) is to run the software module on that node.

4.2.3 Dynamic Resources Allocation Mechanism

After initially deploying the application modules, a run-time policy to balance the workload (in order to keep the energy efficiency high) is needed. To this end, software module migration (e.g., migration of containers or VMs) is put in place. One of the issues related to migration is that migrating containers/VMs
one by one is not leading to the optimal balance, and thus to a good energy efficiency. Such kind of adaptation is more related to an optimization problem known as the bin-packing problem. It can be formulated as following: given a set of objects (in our case Containers/VMs) which are characterized by a size, and given a set of bins (in our case the set of compute nodes) characterized by a capacity, it is required to allocate the objects in such way the capacity of the bins is not exceeded and a given cost function is minimized. The problem has been demonstrated to be NP-hard, thus a heuristic solution is required for practical uses. Among different heuristics, evolutionary approaches provided good results in a reasonable time. Among different evolutionary algorithms, particle-swarm optimization can provide well stable solutions, as well as it showed to converge faster than other approaches (e.g., genetic algorithms, ant colony, particle swarm optimization, etc.).
In order to validate this solution, a testbed hosted in the facilities of CSI PIEMONTE has been set up. In this section an overview and the main features of this environment are reported, describing the architecture and the resources involved.

![Diagram](image)

**Figure 12 - Testbed setup with 3 physical nodes configured to run OpenStack components, and hosting VMs and Containers.**

The testbed infrastructure is represented in Figure 12, and it is composed of three physical nodes:

- **Node-1**: mainly run a set of virtual machines each hosting a component of the OpenStack system. Our setup consists of the following modules:
  - Nova Controller manages the lifecycle of compute instances, specifically it is responsible for spawning, scheduling and decommissioning of machines (virtual machines and LXC containers) on demand.
  - Heat orchestrates multiple composite cloud applications by using a starting template and a dedicated set of APIs.
  - Neutron enables network connectivity as a service for other modules (services).
  - Ceilometer monitors and meters the infrastructure.
  - Cinder provides persistent block storage to running instances.
  - Keystone provides an authentication and authorization service for other OpenStack services, and it provides a catalogue of endpoints for all OpenStack services.
  - Glance stores and retrieves virtual machine and container images which are used by the Nova module for instantiating VMs and containers.
  - Magnum is an OpenStack API service developed to orchestrate containers engines such as Docker.

- **Node-2**: is used to host virtual machines through the KVM hypervisor.

- **Node-3**: is used to host LXC containers through the LXC/LXD hypervisor.

The basic working mechanism is that Heat component receives a template describing how the application is organized, and through the API provided by the Nova module it is able to instantiate VMs/Containers with the specific application service/component installed (e.g., a database). To this end, it checks the availability of the specific images before instantiating the VM or container (it is of worth to highlight that during the first validation experiments –the VDI use case– only virtual machines managed through KVM will be used). Docker containers can be launched both on top of LXC containers and VMs. In the presented installation (using CSI-Piemonte machines), OpenStack modules are hosted within virtual machines running on the same physical node (node-1). While the Magnum component can drive the instantiation of Docker containers in the compute nodes, in OPERA we plan to instantiate Docker containers on top of the VMs/LXC containers. So, in the first phase of the project we are planning to not directly use Docker containers, while opting for LXC version.
6 WORKLOAD MIGRATION

The dynamic workload migration requires ability to migrate the virtual machines and containers that execute the applications. While migration of virtual machines in heterogeneous environment is possible using binary translation, the overhead of running a VM on different ISA will be tremendous. Hence, we limit workload migration only to applications that run in the containers.

The existing implementations of container engines in Linux rely on CRIU tool for checkpoint, restore and migration of the containers. CRIU provides three interfaces to perform these operations: command line (CLI), remote procedure call (RPC) and a C library. The most comprehensive interface is the CLI as the RPC and library interface implement only partial support for features available in CRIU.

To achieve low downtime of the migrated applications and perceive their responsiveness, we extend CRIU with post-copy memory migration for the Linux containers. The post-copy memory migration relies on the Linux kernel mechanism called `userfaultfd`.

6.1 COMPARING POST-COPY AND PRE-COPY MIGRATION APPROACHES

The container state snapshot contains lots of components that describe process state, open file descriptors, sockets, Linux namespaces, state of virtual and pseudo devices. Yet, all these objects are small and can be easily migrated between different hosts with negligible latency. The part of the container state requiring most of the storage capacity for a snapshot or most of the network bandwidth for a migration is the memory dump if the processes that run in a container.

For the case of the container migration, amount of the memory used by the applications running inside the container defines the time required to migrate the container, as well as the downtime of the application. The simplest and naive implementation of container migration is as follows:

- Freeze all the processes inside the container;
- Create a snapshot of the entire container state, including complete memory dumps of all the processes;
- Transfer the container state snapshot to the destination node;
- Restore the container from the snapshot.

In this case, the time required to migrate the container and the downtime of the application running inside it are equal and both these times are proportional to the amount of memory used by the processes comprising the container.

The application downtime during migration may be decreased with one or more round of memory pre-copy before freezing the container. In this case the container migration algorithm outline is:

- Create a memory dump for all the processes running inside the container;
- Enable tracking for memory changes for these processes;
- Transfer memory dump to the destination;
- If amount of memory changes is below some predefined limit:
  - Freeze all the processes inside the container;
  - Create a snapshot of the entire container state, the memory dump contains only modified parts;
  - Transfer the state snapshot to the destination;
  - Restore on the destination;
- Otherwise:
  - Create a dump of modified memory;
  - Continue to check the amount of the memory changes.

With iterative memory pre-copy container migration time is slightly longer that in the simple case, but the actual downtime of the application is significantly smaller in most cases.

However, such approach may not work for applications with rapidly changing memory working set. For such applications, the amount of modified memory will always be higher than the desired threshold and therefore the iterative pre-copy algorithm will never converge.
Another approach for reduction of application downtime is called "post-copy migration". In this approach, the memory dump is not created and memory contents is transferred after the application is resumed on the destination node. The post-copy container migration algorithm is outlined below:

- Freeze all the processes inside the container;
- Create a snapshot of the entire container state, except memory dumps of all the processes;
- Transfer the container state snapshot to the destination node;
- Restore the container from the snapshot;
- Transfer the memory contents of the processes on-demand according to the memory accesses of the running processes;
- Transfer the contents of the memory that was not yet explicitly accessed in the background.

The primary advantage of post-copy migration is its guaranteed convergence. Additionally, post-copy migration requires less network bandwidth that iterative pre-copy migration because the memory contents is transferred exactly one time. The migration time in this case is small because only the minimal container state snapshot is transferred before the execution is resumed on the destination node. The application downtime is almost as small as the migration time, however, immediately after migration the application will be less responsive because of the increased latency for memory access.

6.1.1 Userfaultfd

The userfaultfd mechanism in the Linux kernel allows user-level implementation of demand paging. The userfaultfd mechanism provides two interfaces: `userfaultfd` system call and `ioctl`'s specific to the userfaultfd.

The `userfaultfd()` system call creates a new userfaultfd object that can be used for delegation of page-fault handling to a user-space application, and returns a file descriptor that refers to the new object. The new userfaultfd object is configured using `ioctl` system call. Once the userfaultfd object is configured, the application can use `read` system call to receive userfaultfd notifications.

Various `ioctl()` operations can be performed on a userfaultfd object. Some of these operations are used to userfaultfd behavior. They allow the caller to choose what features will be enabled and what kinds of events will be delivered to the application. There are also operations that enable the calling application to resolve page-fault events.

6.2 CRIU COMMAND LINE INTERFACE

CRIU is a single executable that supports several actions. The most important actions are `dump` and `restore` that create a checkpoint of a set of running processes and restore a set of running processes from previously saved checkpoint. With addition of helper actions, such as `pre-dump`, `page-server` and `lazy-pages`, CRIU can implement live migration of Linux containers.

The CRIU actions accept variety of options that allow the user to specify where the checkpoint dump will be stored, what should be the log verbosity and how different properties of the Linux containers, such as namespaces, mounts and cgroups should be treated.

The detailed list of CRIU command line options can be found at [https://criu.org/CLI](https://criu.org/CLI).

6.3 CRIU MIGRATION PROCESS

CRIU migration process consists of several steps:

- Freeze the applications of the source node
- Save the container state into the image files
- Copy the state to the destination node
- Restore the container on the destination

CRIU allows iterative dumps of the applications memory without stopping the container on the source node, thus enabling minimization of memory transfer during the application freeze.

During the OPERA project, we’ve implemented post-copy or lazy restore support in CRIU that allows further optimization of the application down-time and network bandwidth used for the container migration.
Below is an example of commands that can be used for post-copy migration of a process tree:

```
src # criu dump --images-dir /path/to/checkpoint --tree $(pidof my-app) --lazy-pages --port 9876
src # rsync -aP /path/to/checkpoint dst:/path/to/checkpoint
dst # criu lazy-pages --images-dir /path/to/checkpoint --page-server --address dst --port 9876 &
dst # criu restore --images-dir /path/to/checkpoint --lazy-pages
```

6.4 LXC CONTAINER MIGRATION

LXC container hypervisor provides a convenience wrapper to the CRIU tool. This wrapper allows seamless migration of LXC containers from one compute node to another. In order to migrate a container, all the user has to do is to invoke the following command:

```
lxc move h1:c1 h2:
```

where h1 is the hostname of the source node, h2 is the hostname of the destination node and c1 is the name of the container.
7 CONCLUSIONS

This document presents a summary of the work carried out in work package 5 (WP5) - task T5.3. The main objective of such research activity is to define the interface and the way of integrating the ECRAE system (see D5.4) with the cloud infrastructure and management software, that is represented by OpenStack. Specifically, we studied how to leverage on the modularity of the OpenStack environment to provide such integration with the software component that we are designing and developing in T5.4. Also, the interface with the CRIU Linux container management system has been considered and analyzed, along with the mechanism to migrate containers.

Initially, this document draws the main motivation behind the use of a cloud orchestration. This discussion is important to understand the requirements that cloud services providers have with respect to the management of their entire infrastructure. Following this line, the selected cloud management environment (OpenStack) has been deeply studied and discussed. OpenStack provides a modular solution to the orchestration problem: different modules can be enabled and integrated in order to provide specific management services, as well as to facilitate the monitoring of the infrastructure. To this end, the understanding of their interfaces and the interactions among them is fundamental to properly integrate external components. From this viewpoint, the main OpenStack modules we decided to enable are described. Another important point discussed in this document regards “flavours”. Flavours are the mean for describing the configuration of a certain resource (generally a node), that is required to run a specific service or software component. Flavours, are also used in ECRAE (see D5.4) to select the most appropriate node where to launch the execution of a specific software component of the cloud application. To this end, flavours are analyzed also in relationship with the Knowledge Base (KB), i.e., the database used by the ECRAE to rank the nodes and select the most effective one.

Starting from these information, the we described the orchestration toolchain used to deploy cloud applications in the data center. Such toolchain is based on the integration of the ECRAE module, which is responsible, for each application component, to select the most appropriate execution node. Finally, the discussion on the workload migration leveraging on the CRIU system is presented.

Future activities related to T5.3 will be focused on the integration of different infrastructural components together and to analyze their actual interaction.
8 REFERENCES


[5] RED HAT OPENSTACK PLATFORM, ARCHITECTURE GUIDE

https://docs.openstack.org/admin-guide/common/get-started-with-openstack.html

[7] Userfaultfd (1)
https://www.kernel.org/doc/Documentation/vm/userfaultfd.txt

[8] Userfaultfd (2)
http://man7.org/linux/man-pages/man2/userfaultfd.2.html

[9] Userfaultfd (3)

[10] CRIU (1)
https://criu.org/CLI

[11] CRIU (2)
https://criu.org/Userfaultfd

[12] CRIU (3)
https://criu.org/Lazy_migration
