



ACTiCLOUD: ACTivating resource efficiency and large databases in the  
CLOUD

Project No: 732366

H2020-ICT-2016-1

D2.3: Rack scale MicroVisor v2.0

<b>Due date of deliverable:</b>	<b>M32 (2019/08/31)</b>
Actual submission date:	M33 (2019/09/13)

**Executive summary:**

Deliverable D2.3 provides the final version of ACTiCLOUD's rack-scale Hypervisor. The deliverable consists of software (binaries and source code), as well as the current documentation. This deliverable is an updated version of Deliverable D2.1, which presented the initial implementation of the rack-scale Hypervisor.

**List of authors:**

Author	Affiliation
Michail Flouris	OnApp
Stelios Louloudakis	OnApp

Dissemination Level	<input checked="" type="checkbox"/>	<b>PU (Public)</b>
	<input type="checkbox"/>	PP (Restricted to other programme participants)
	<input type="checkbox"/>	RE (Restricted to a group specified by the consortium)
	<input type="checkbox"/>	CO (Confidential, only for members of the consortium)
	Where restricted, access granted to:	
Nature	<input type="checkbox"/>	R (Report)
	<input type="checkbox"/>	P (Prototype)
	<input type="checkbox"/>	D (Demonstrator)
	<input checked="" type="checkbox"/>	<b>O (Other)</b>

Review Status	<input type="checkbox"/>	Draft
	<input type="checkbox"/>	WP Leader accepted
	<input type="checkbox"/>	QA approved
	<input checked="" type="checkbox"/>	<b>Coordinator accepted</b>

**Revision History:**

Version	Author(s) (Affiliation)	Notes
0.1	Michail Flouris (ONAPP)	Initial ToC
0.2	Michail Flouris (ONAPP) Stelios Louloudakis (ONAPP)	Initial draft content
0.5	All authors	Final version for internal review
0.7	George Goumas (ICCS) Atle Vesterkjær (Numascale)	Deliverable reviewed
0.9	All authors	Addressing reviewers comments
1.0	All authors	Final version
1.1	Vasileios Karakostas	Submitted version

**ACTiCLOUD Consortium:**

Participant No	Participant organisation name	Short name	Country
1 (Coordinator)	Institute of Communication and Computer Systems	ICCS	Greece
2	Numascale AS	NSCALE	Norway
3	Kaleao Limited	KALEAO	UK
4	OnApp Limited	ONAPP	Gibraltar
5	University of Manchester	UNIMAN	UK
6	MonetDB Solutions B.V.	MDBS	Netherlands
7	Neo Technology	NEO	Sweden
8	UMEA University	UMU	Sweden



NUMASCALE

KALEAO

onapp



**Confidentiality:**

This document contains proprietary and confidential material of certain ACTiCLOUD contractors, and may not be reproduced, copied, or disclosed without appropriate permission. The commercial use of any information contained in this document may require a license from the proprietor of that information.

THIS DOCUMENT IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES

(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENT, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## Table of Contents

<b>1</b>	<b>Introduction .....</b>	<b>9</b>
1.1	Purpose of this Document .....	9
1.2	Relevance to ACTiCLOUD objectives, business scenarios and use cases .....	9
1.3	Document Structure .....	10
<b>2</b>	<b>The Rack-scale MicroVisor architecture .....</b>	<b>11</b>
2.1	Virtualization and traditional Hypervisors.....	11
2.2	Motivation for the MicroVisor Hypervisor .....	12
2.3	Enhancements of the MicroVisor architecture .....	12
2.4	The Rack-scale Hypervisor .....	13
2.4.1	Distributed Storage Platform.....	15
2.4.2	Guest Operating System support.....	15
2.5	Reliability and Availability .....	15
2.5.1	Storage Fault-tolerance & Availability .....	16
2.5.2	High-Availability Support for the MicroVisor management layer .....	16
2.6	Rack-scale Monitoring .....	17
2.6.1	Monitoring Statistics.....	17
2.6.2	Performance counters on Virtualization.....	19
2.7	Performance.....	20
2.7.1	vNUMA Support.....	20
2.8	Web-based User interface.....	24
<b>3</b>	<b>Resource control in the MicroVisor platform .....</b>	<b>30</b>
3.1	Relevance to ACTiCLOUD Objectives .....	30
3.2	Performance features of Resource Groups .....	30
3.2.1	NUMA CPU assignment.....	30
3.2.2	CPU overcommit.....	31
3.2.3	CPU Pinning (Core Assignment) .....	31
3.2.4	Storage Optimisation .....	31
3.3	Reliability and availability features of Resource Groups.....	31
3.3.1	Fault-tolerance.....	32
3.3.2	Failover .....	32
3.4	Configuration of Resource groups via the UI.....	32
3.4.1	Command-line support for resource groups.....	37
3.4.2	API support for Resource groups (resourceGroupsPost) .....	38

3.5	Storage & Configuration of Datastore .....	41
3.5.1	Command-line support for datastores.....	44
3.5.2	API support for Datastores (datastoresPost).....	45
<b>4</b>	<b>Integration of the MicroVisor into OpenStack.....</b>	<b>50</b>
4.1	Managing rack-scale resources through OpenStack.....	50
4.2	Revised OpenStack architecture on the MicroVisor platform.....	51
4.2.1	Implementation of OpenStack Pike in the previous prototypes .....	51
4.2.2	A new OpenStack management architecture for the Final prototype.....	53
4.3	OpenStack Implementation for the Final Prototype .....	53
4.3.1	External API.....	54
4.3.2	OpenStack drivers .....	55
4.3.3	Network: Neutron.....	55
4.3.4	Installation scripts.....	56
4.3.5	Open-source release of OpenStack drivers.....	56
4.4	Control via OpenStack's Horizon UI .....	56
4.4.1	Listing Hypervisors in the MicroVisor cluster.....	57
4.4.2	Creating Instances .....	57
4.4.3	Creating Flavors.....	61
<b>5</b>	<b>Conclusions.....</b>	<b>62</b>
<b>6</b>	<b>Appendix I .....</b>	<b>63</b>
6.1	MicroVisor External API for OpenStack Pike .....	63
6.3	Hypervisor Performance counters .....	82

## List of Figures

Figure 1: Memory throughput .....	24
Figure 2: MicroVisor UI – Dashboard showing platform resources .....	25
Figure 3: MicroVisor UI – Rack view showing their resources .....	26
Figure 4: MicroVisor UI – Resource group view showing grouped resources .....	27
Figure 5: MicroVisor UI – Datastore view showing configured datastores and used storage capacity.....	28
Figure 6: MicroVisor UI – Instance view showing virtual machines configured and deployed .....	29
Figure 7: Resource group configuration .....	33
Figure 8: Storage optimization .....	34
Figure 9: CPU overcommit.....	34
Figure 10: Fault tolerance.....	35
Figure 11: Datastores selection .....	35
Figure 12: Nodes selection.....	35
Figure 13: Cores selection.....	36
Figure 14: Networks selection.....	36
Figure 15: RG configuration finalization .....	36
Figure 16: RG availability.....	37
Figure 17: Creating a datastore.....	41
Figure 18: Compute nodes selection .....	42
Figure 19: Physical disks selection .....	42
Figure 20: Datastores redundancy.....	43
Figure 21: Enable metadata .....	43
Figure 22: Finalizing Datastore .....	43
Figure 23: Created Datastores list.....	44
Figure 24: A high-level view of OpenStack core services. ....	50
Figure 25: OpenStack - MicroVisor Architecture .....	52
Figure 26: New architecture for OpenStack on the MicroVisor platform .....	53
Figure 27: List of Hypervisors .....	57
Figure 28: Create instance details .....	58
Figure 29: Create instance source.....	58
Figure 30: Create instance flavor.....	59
Figure 31: Create instance networks.....	59
Figure 32: Create instance key pair .....	60
Figure 33: New instance created.....	60
Figure 34: Create flavor form.....	61

**List of Abbreviations**

Abbreviation / Acronym	Meaning
DoA	Description of Action
WP	Work Package
NFV	Network Function Virtualization
VM	Virtual Machine
OS	Operating System
SPOF	Single Point Of Failure
I/O	Input / Output
Dom0	Domain 0 (Control Domain)
DomU	User domain
VMM	Virtual Machine Monitor
KVM	Kernel-based Virtual Machine
HVM	Hardware Virtual Machine (fully virtualized hardware)
CSP	Cloud Service Provider
PV	Para-Virtualization
LUN	Logical UNit (virtual or physical disk volume)
DB	DataBase
WP	Work Package
WPL	Work Package Leader



# 1 Introduction

ACTiCLOUD's vision is to develop a novel cloud architecture that will break the existing scale-up and share-nothing barriers and enable the holistic management of physical resources both at the local cloud site and the distributed levels, targeting drastically improved utilization and scalability of resources. This will ultimately translate to:

- a) Significant cost and performance improvements for Cloud Service Providers (CSPs)
- b) Higher performance stability and lower pricing for cloud applications
- c) Enhanced flexibility and scalability of cloud resources for intensive database applications that have until now faced tough challenges in covering their resource demands from existing cloud offerings.

ACTiCLOUD aims to advance the business viability of cloud deployment scenarios through increased resource utilization and flexibility. This goal will be achieved through enhancement of the various technologies in the architecture components, i.e. the Hypervisor, the cloud manager, system libraries, language runtimes and database systems with a novel and holistic set of mechanisms and policies built on top of these new-generation computing system architectures and therefore enabling distributed, hyper-converged, "share-anything", resource scale-out cloud platforms to broaden the applicability of cloud technology across more markets through richer and more cost effective application deployments.

## 1.1 Purpose of this Document

This document shows the valuable contribution of the MicroVisor Hypervisor in ACTiCLOUD. It is the report/documentation part of Deliverable D2.3, accompanying the updated software part of the deliverable that contains software components (binaries and code) for the MicroVisor Hypervisor platform and the components required for the integration with the ACTiManager.

The MicroVisor platform, as ACTiCLOUD's rack-scale Hypervisor layer, is a significant component of the platform, providing all necessary mechanisms for virtualizing, managing and monitoring compute, network and storage resource across the rack, as well as the mechanisms to reconfigure resources on demand, according to the ACTiManager distributed cloud resource manager policies (discussed in Deliverable D2.1) to achieve increased resource efficiency across the ACTiCLOUD platform.

## 1.2 Relevance to ACTiCLOUD objectives, business scenarios and use cases

MicroVisor is one of the most important components in the ACTiCLOUD architecture as it plays a central role in the realization of ACTiCLOUD's objectives towards next generation IaaS platforms [ACTi\_D1.1]. The MicroVisor platform, through the efficient Hypervisor and the OpenStack integration, enhances state-of-the-art cloud management and provides efficient and effective resource control approaches with mechanisms to:

- drastically increase the resource efficiency of cloud infrastructures in terms of throughput per resource (Strategic Objective S01.1 on resource efficiency),
- provide applications with better performance by lowering virtualization overheads and strict performance guarantees through better resource allocation and placement (Strategic Objective S01.2 on performance stability).

Additionally, as part of the ACTiCLOUD architecture, the rack-scale MicroVisor supports all the necessary mechanisms for the deployment and management of scale-up and scale-out resources offered by the underlying hardware (Strategic Objective S02.1 on scalability in resource provisioning). Furthermore, interacting with the ACTiManager and the applications running in Virtual Machines (VMs), it allows placement and scheduling of VM resources on the hardware and responds to dynamic resource requests of the ACTiManager to address changes in application demand (Strategic Objective S02.2 on elasticity in resource provisioning).

Finally, the efficient rack-scale resource management support of the MicroVisor is an essential component for the realization of ACTiCLOUD's business scenarios [ACTi\_D1.1, ACTi\_D1.2], also summarized below:

- Business scenario 1: Effective consolidation for increased revenue and reduced TCO,
- Business scenario 2: Workload prioritization,
- Business scenario 3: Hosting larger workloads,
- Business scenario 4: Collaboration with sibling cloud sites,
- Business scenario 5: Enhanced dependability and availability.

### 1.3 Document Structure

This deliverable contains the following sections: Section 2 presents the MicroVisor platform, including a general overview of virtualization, traditional Hypervisor architectures and the motivation for building and using the MicroVisor within the ACTiCLOUD project. Section 2 also discusses the enhancements the MicroVisor provides, including the reliability, availability and resiliency features that are integrated in the design of the platform, the performance counters used for measuring the performance of the MicroVisor, as well as the NUMA-awareness and vNUMA support implemented. All these features are directly relevant to the ACTiCLOUD project objectives and the integration with management components of the ACTiManager.

Section 3 analyses the resource control capabilities that are built in the MicroVisor platform, including the concept of resource groups, as well as their performance, availability and reliability features. Section 3 also describes the process of configuring datastores and resource groups with these features. In section 4, the integration of the MicroVisor into OpenStack is described, including the redesigned architecture, implemented through an external API, along with OpenStack drivers that will be released as open-source. Section 4 also demonstrates the integrated control that can be performed through the OpenStack's Horizon UI. Finally, the Appendixes present some more detailed technical documentation, including the performance counters implemented for the MicroVisor and a more detailed reference of the external REST API developed for the integration with OpenStack Pike drivers.

## 2 The Rack-scale MicroVisor architecture

The MicroVisor platform, as ACTiCLOUD's rack-scale Hypervisor layer, is a significant component of the platform, providing all necessary mechanisms for virtualizing, managing and monitoring compute, network and storage resource across the rack. In this section we provide a short introduction to virtualization and an overview of the MicroVisor platform and its concepts.

### 2.1 Virtualization and traditional Hypervisors

Virtualization of server hardware is a commonly used practice to provide scalable resource management and it is essentially the enabling technology for cloud computing. There are many benefits of virtualizing hardware resources, primarily to enable efficient resource sharing (CPU, memory, NICs) across a multi-tenant platform hosting a variety of Operating Systems (OSes). A Hypervisor (HV), or virtual machine monitor (VMM) is a piece of software that creates, manages and runs Virtual Machines (VMs), or guest domains. The Hypervisor presents the guest operating systems with a virtual operating platform and manages the execution of the guest operating systems. In addition to commonly deployed commercial Hypervisors such as VMware, there are two dominant open-source Hypervisor platforms: Kernel-based Virtual Machine (KVM)<sup>1</sup> and the Xen Hypervisor<sup>2</sup>.

The Xen Hypervisor provides a true Type I Hypervisor, in contrast to the KVM Hypervisor platform. That is to say that the Hypervisor layer runs directly on the bare-metal hardware, managing guest OS instances directly above it. There is not any host operating system required and in this way the Type I architecture is considered to be a minimal, high performance shim. In parallel to the virtualized guest systems running on the Type I Hypervisor, traditional Xen systems utilize a Control domain, known as Dom0, which has privileged access to the Hypervisor and is responsible for various tasks including; administering guest virtual machines, managing resource allocation for guests, providing drivers for directly attached hardware, and offering network communication support. Guest Virtual Machines (DomUs) do not typically have direct access to real hardware, and thus all guest domain network and storage communication is managed through para-virtual device drivers hosted in the Control domain (Dom0), which in turn handles safe resource access through multiplexing the physical hardware.

For every guest VM para-virtualized (PV) device that is active, there is a corresponding driver in the control domain that allocates resources and handles the communication ring over which virtual IO requests are transmitted. The control domain is then responsible for mapping those IO requests onto the physical hardware devices behind them. Based on the Xen Hypervisor platform, a guest OS can access a paravirtual IO interface using the standard paravirtualized (PV) driver such as Netfront for the Ethernet or Blkfront for the block storage device without any actual underlying hardware knowledge. In contrast to PV, HVM guests are fully virtualized, providing hardware support through their own unmodified drivers in the VM, while the hypervisor provides full support for hardware devices, either native or through hardware emulation.

---

<sup>1</sup> Kernel Virtual Machine <https://www.linux-kvm.org/>

<sup>2</sup> Xen Project <https://www.xenproject.org/>

## 2.2 Motivation for the MicroVisor Hypervisor

Considering Xen as a reference Type I hypervisor, we observe that its architecture for IO (storage and network requests), is quite centralized, where sending all the IO requests from guest domains through a central, core domain, Dom-0, itself becomes a bottleneck that affects performance. Thus, the network and I/O performance in this architecture is limited, due to context switches to the control domain (Dom0) that handles the network packet switching for all VMs. In addition, network function virtualization is increasingly being handled on commodity hardware as individual worker VMs to provide enhanced network packet filtering, middlebox functionality and packet forwarding. A new model is required to enhance and advance this architecture by making the packet forwarding and I/O layers as fast as possible, with slow path network functions offloaded to separate processing engines (Software Defined Network and Network Function Virtualisation model).

To address these limitations of a traditional Type I hypervisor architecture (e.g. Xen), OnApp has developed a new optimized hypervisor platform named the “MicroVisor”. The MicroVisor implementation within the framework of ACTiCLOUD removes the centralised Dom0 model of a Type-1 hypervisor and instead passes the logic and control to a higher-level software tool that can then communicate with the guests. The core idea behind this approach is to remove any dependency on a local control domain (Dom0) for virtual machine setup, booting and resource allocation, and instead move this generic functionality into the hypervisor layer itself.

To ensure that control logic is handled efficiently and quickly in the MicroVisor, raw Ethernet frames are sent among guests, avoiding TCP/IP overhead, which has a profound effect when sending and receiving low-size block requests across the network. One of the issues of moving to a controller-less approach is that this means that the drivers, which were part of the control domain (Dom0), now need to be moved to another part of the Hypervisor. In the MicroVisor this is implemented by either:

1. Creating driver domains that are lightweight Mini-OS<sup>3</sup> style domains that have access to the drivers and the underlying hardware. Each piece of hardware will have at least one driver domain associated with it, with which the guests communicate for accessing the physical resources.
2. Integrating a device driver for the hardware in the hypervisor layer itself. This has even lower overhead for IO devices, however it is more complex and costly to implement, so only a few select drivers have been implemented currently in the MicroVisor.

## 2.3 Enhancements of the MicroVisor architecture

This section describes the advancements of the MicroVisor approach, over other traditional hypervisor architectures, such as Xen. The main difference of the MicroVisor from Xen and KVM is that there is no control domain (Dom0 in Xen terminology, or host OS for KVM). The centralised Dom0 model means that split driver requests have to pass through the control domain for each request and it becomes a bottleneck when many requests are being served to multiple guest domains (DomUs). In order to interface with hardware, which is one of the normal roles of the control domain in this new architecture, new driver domains are needed. These driver domains interface with the hardware and in the current implementation of the MicroVisor

---

<sup>3</sup> <https://wiki.xenproject.org/wiki/Mini-OS>

are implemented through minimal Linux domains or via Mini-OS domains for adding the driver support. To provide even higher performance, in a few special cases (e.g. some network interfaces or NVMe drives), such drivers have been integrated in the hypervisor itself.

Due to the absence of a control domain (Dom0), the MicroVisor architecture handles configuration and management significantly different from traditional hypervisors, such as Xen or KVM, which can use Dom0 (or hostOS in KVM) for configuration and management services. To allow efficient management, the MicroVisor architecture has moved a minimal essential set of tools from Dom0 to the Hypervisor layer itself, adding support for control and configuration APIs needed.

In addition to moving configuration and control from Dom0 to the Hypervisor, the MicroVisor architecture features a remotely accessible (i.e. networked) control and management API, to accommodate the fact that there is no control domain to run locally on each MicroVisor. The result is an Ethernet-level command API that allows location-agnostic monitoring and control of any Hypervisor from a controller anywhere on the local Ethernet network. The MicroVisor Ethernet-level API can be used in a distributed mode for monitoring and managing resources on multiple remote Hypervisors (compute, network, and storage resources) from a single controller, running on a VM anywhere in the local cluster.

Another important characteristic of the MicroVisor architecture is that storage I/O requests cannot be sent in the usual block-back queues through Dom0 to storage drives, since there is no Dom0. Instead the MicroVisor architecture, being designed specifically for distributed operation, converts the I/O requests into Ethernet frames and sends these directly via the ATA-over-Ethernet protocol to either the local or remote storage server backends. Using Ethernet frames ensures that the overhead of TCP/IP processing is removed and as such the I/O virtualization overhead is significantly reduced. Overall, the use of Ethernet control frames for the management, control, I/O and monitoring of the MicroVisor platform, along with the driver domain and removal of Dom0 ensures that the virtualization overhead is much lower than the standard hypervisor implementations. This results in higher performance and better control for each of the resources and the guest domains.

Finally, an additional important aspect is that the MicroVisor platform is designed to be a lightweight Hypervisor platform that is better suited for emerging hardware platforms compared to traditional Hypervisors (e.g. Xen or KVM). Currently x86 (Intel/AMD) platforms have used NUMA architectures for increasing the number of processor sockets and addressable memory on a single board. Cores in NUMA hardware are linked with certain regions of memory but can access the entire system memory at a higher incurred latency cost and at lower speeds. ARM based, micro-server hardware platforms incorporate many cores that have lower performance than the more widely used x86 (Intel/AMD) server platforms but are more energy efficient, and systems like Kaleao's KMAX are entering the data center server market as a cost effective alternative. The MicroVisor has been designed to work on both x86 (Intel/AMD) and ARM hardware platforms, but MicroVisor's performance and power advantages are especially pronounced in the aforementioned low-power ARM-based processors that have fewer resources.

## 2.4 The Rack-scale Hypervisor

The ACTiCLOUD architecture subcomponents of the Hypervisor platform are listed below, including a brief description of each component and its functionality related to the specific technology embraced i.e., the MicroVisor platform.

The MicroVisor hypervisor layer consists of the following subcomponents:

- **Hypervisor microkernel.** The main Hypervisor microkernel is like Xen (Type I), but with significant differences in the handling of networking, I/O, and resource management. Additionally it has a new Ethernet-based management layer and several more features, detailed in the rest of this list.
- **Resource scheduler.** The resource scheduler is part of the Hypervisor kernel and can be configured through the MicroVisor Ethernet-level API. Pinning of resources can be carried out accordingly. The scheduler then decides how the resources are presented through the driver domains to the guest Virtual Machines and so can be used for rate-limiting and performance configuration.
- **Virtual Switch.** An integrated internal packet switch is embedded in the Hypervisor kernel to handle packet forwarding. This switch manages shared network and I/O resources from different MicroVisor nodes to provide aggregated network and storage resources that are linked and accessed from the guest VMs on each Hypervisor.
- **Ethernet packet handler.** The MicroVisor does not use TCP/IP for management or resource control, using instead Ethernet packets. This avoids having the heavy TCP/IP stack within each domain, but this does mean that a network reliability and flow control protocol has to be implemented for moving control and monitoring data across Hypervisors. This, however, is much lighter than TCP. VMs residing on the MicroVisor can continue to use TCP/IP.
- **Driver domains and integrated drivers.** For the MicroVisor, given that there is no control domain (no Dom0), a driver when required is launched as its own Virtual Machine (driver domain) that has access to the physical resource that can then be shared with the guests. The current implementation of the driver domain uses MiniOS as the host OS, which is a cut down minimalistic OS kernel used for stub domains. Additionally the MicroVisor has integrated a small set of hardware drivers in the hypervisor layer itself, in order to further optimize performance on specific network cards and NVMe storage drives.
- **Monitoring system.** A monitoring system is built into the MicroVisor, also through Ethernet packets. Some values that would normally be exposed in GNU/Linux via the “/proc” interface, for instance, are exposed through the Monitoring API and captured by aggregators on the local network. Currently, several statistics about CPU/network/memory/storage usage are captured and can be queried. Depending on the requirements of the higher level components, these monitoring metrics can be extended.
- **Monitoring API.** The monitoring values captured by aggregators on the controller node are stored using the round robin database (RRD) format. Through a monitoring API provided, one can access the current status of a particular MicroVisor. The API describes the resource that is being queried and how frequently the monitoring should be carried out.
- **Orchestration API.** Control of the MicroVisor is carried out through the MicroVisor API. Assigning workloads, managing VMs and the connected resources is carried out through this API.

In addition to the Hypervisor layer, the MicroVisor platform includes several other components that provide essential functions. These are described in the following sections.



### 2.4.1 Distributed Storage Platform

In the MicroVisor platform, besides the Hypervisor itself, OnApp has developed a distributed, hyper-converged storage platform that is currently deployed in the OnApp Cloud platform as a separate storage product. OnApp Storage allows for Hypervisors to use directly-attached storage on the server machines as a distributed block-storage system. This has been used for many years with Xen and KVM and is used in a large part of the ONAPP customer base. It is also considered as a disruptive technology versus the conventional network attached storage (NAS) platforms that are usually used in the data center. Most of the storage platform has been adapted from the OnApp Storage solution for the MicroVisor platform, such that it can be used to share storage resources on clusters of machines running the MicroVisor platform. More details on the storage layer implementation with a focus on the reliability and availability features is provided in Section 2.5.1 “Storage Fault-tolerance & Availability”.

### 2.4.2 Guest Operating System support

The Guest Operating System (OS) refers to the typical OS that is inside a virtual machine (VM). The requirements for the Guest OS within the ACTiCLOUD project are to support unmodified OSes for guest domains on established cloud infrastructures, which is exactly what the MicroVisor platform supports. Thus, ACTiCLOUD-enabled systems will be able to operate with any type of Guest OS, although our experimentation and any changes to the system libraries of the Guest OS is performed on established Linux-based systems. Hence, within the scope of ACTiCLOUD, we focus our effort on Linux-based systems, and particularly on the most popular Linux-based cloud image distributions (e.g. Ubuntu 16.04, 18.04, CentOS 7, etc.) for the final project prototype.

To create and deploy a new VM with a guest OS (e.g. Ubuntu 18.04), the user needs to use the standard OpenStack User Interface (UI), called Horizon UI, or the OpenStack command line interface (CLI) that controls the MicroVisor platform. The basic steps of this process are the following:

1. The image of the target OS that will be used for the instance (VM) is downloaded,
2. The user specifies through the OpenStack UI or CLI a flavor that will provide the specifications for the instance (according to OpenStack terminology), as well as the network and storage settings for the instance,
3. The instance (VM) is created and the Guest OS starts its execution.

More information about this process can be found in the accompanying software part of this deliverable and in the OpenStack documentation (Pike version<sup>4</sup>).

## 2.5 Reliability and Availability

ACTiCLOUD aggregates resource pools and, as such, may increase the likelihood for failures on part of the system that affect other applications. Graceful degradation of the shared resources is important to avoid the case where failures have an effect on greater parts of the system.

Since CSPs usually have redundancy in hardware, the MicroVisor storage platform offers increased availability and reliability by utilizing redundant resources through data replication. This type of replication is based on redundancy across Hypervisor, so it can maintain availability of a storage volume, not only when one of its storage devices fails, but also when a Hypervisor

---

<sup>4</sup> <https://docs.OpenStack.org/install-guide/launch-instance.html>

that hosts one of the replicas for that virtual disk fails. Based on this level of storage availability, the platform is also able to offer VM failover to another node, since the data volume is available, even if a server fails.

### 2.5.1 Storage Fault-tolerance & Availability

The MicroVisor platform has an integrated distributed block storage service, where VMs running on any of the Hypervisors can access any local or remote virtual volume through block I/O requests that the MicroVisor converts to ATA-over-Ethernet requests to local or remote backend I/O servers. The backend storage pool consists of specialized virtual machines (VMs), called storage nodes that have privileged access to the physical storage devices (e.g. hard disk drives, SSDs or NVMe drives) that reside on each server machine. Each storage node is responsible for handling the block I/O requests of the drives it is directly managing.

A physical storage device can only be associated with one storage node. However, the platform offers storage pools, called *Datastores* (configuration details analyzed in Section 3), which are logical layer configurations that map the physical storage drives into distributed storage pools with different properties (e.g. replicas for fault-tolerance, or overcommit).

The number of replicas determines how many replicas of a block of data should be created in a pool, and the corresponding level of faults that a virtual storage volume on the datastore can tolerate. This can currently be set to 1 (i.e. one data replica / no redundancy), or 2, meaning the storage block is replicated to one other drive and another server node, so it is not affected by single-node or drive failures.

The overcommit value sets the value of resources allocated beyond the physical capacity that is physically available in the system. In shared storage platforms although 10GB may be allocated to a customer, only 5GB may be actually used with the rest remaining under-utilised. Across many VMs in a shared pool this wasted capacity might be considerable, thus the overcommit option allows this to be set in order to increase actual utilization. Once the actual storage runs out, the CSP will need to migrate some VM storage content to other drives or other storage pools. To implement overcommit, storage nodes maintain a bitmap of the blocks that have been written along with a tag that records the time of the block updates. Any transaction that is fully committed should update this timestamp accordingly on all storage paths to maintain consistency.

Storage replication and overcommit has been re-designed, implemented and failure-tested during the second period of ACTiCLOUD. The final version developed has been integrated in the final prototype of the ACTiCLOUD platform, providing a high level of storage reliability and availability.

### 2.5.2 High-Availability Support for the MicroVisor management layer

Another aspect of the MicroVisor platform that is highly relevant to business scenario 5 (enhanced reliability), as discussed in Deliverable D2.1, is the high-availability (HA) support for the MicroVisor controller. The controller is designed to maintain the system availability, avoiding the existence of a Single Point Of Failure (SPOF). A SPOF is an individual software or hardware component of the whole system, whose failure could cause downtime or data loss. The system downtime occurs when a service or hardware component, such as a virtual or physical network switch, is unavailable beyond a specified maximum amount of time. Potential data loss could occur on accidental deletion or destruction of data, such as the hard disk failure.



The MicroVisor controller HA is achieved through the support of three main components: the MicroVisor management controller, the monitoring of VMs and the distributed storage service.

1. The MicroVisor management controller, running essential management services, runs on a virtual machine (VM) that can be hosted on any MicroVisor host in the cluster or rack, booting from a virtual Logical Unit (LUN) that is replicated. The LUN is announced via a broadcast protocol that is detected by all MicroVisor hosts attached to the same local Ethernet network. A MicroVisor is designated (either via boot arguments or at runtime) to host the controller node, and that particular MicroVisor then runs a periodic task to ensure that the controller node is booting and running. If it is not running, it is started immediately. The process of failing over the controller is therefore a straightforward mechanism to instruct a new MicroVisor to boot and host the controller in the event that the original host for the controller node has failed. Detecting liveness is achieved by running a quorum of separate nodes across the cluster of MicroVisors and using a consensus protocol to decide if the hosting node has disappeared, followed by a leadership election process to determine the new host that will run the controller.
2. The ongoing monitoring of VM liveness is implemented in the controller node. In the event that a controller determines a MicroVisor is no longer responding and all the VMs are dead, it will inform that storage layer to remove the active LUN mapping of the dead nodes and restart them elsewhere. The selection of which MicroVisor to fail-over each VM is based on the resource group that the VM belongs to and the failure policy configured and availability of resources.
3. The distributed storage layer, discussed above, is based on the mature OnApp Integrated Storage technology. This component provides a complete management stack for storage content and provides storage nodes (the individual hosts that control the direct attached storage drives) to communicate amongst each other and determine liveness of each other's replicas. Storage content that is replicated and strictly managed via an epoch mechanism. Whenever a storage replica is lost from the set, the epoch is incremented to ensure that the non-responding member is forcibly removed from the set. If it comes back online, it will always determine across the set of owners that it has a stale set of data and will force itself into a slave status which means that it must be re-synchronised to ensure it is consistent again before it can be added back to the set.

More details on the operation and implementation of these HA components, liveness tracking of services and fault-tolerance of the controller node and its metadata can be found in Deliverable D2.1.

## 2.6 Rack-scale Monitoring

The MicroVisor platform provides mechanisms for monitoring and performance characterization that are essential for the ACTiManager components of the ACTiCLOUD architecture. These are described in this section.

### 2.6.1 Monitoring Statistics

The MicroVisor platform currently includes monitoring mechanisms that gather statistics for the following resources:

- CPU usage
- Network IO stats

- Block storage IO stats
- Memory utilization

The statistics are transmitted to an RRD aggregator service called statsd, running on a local node, which receives stats via Ethernet packets and converts them to the RRD time-series form. To ensure isolation and separate handling, the MicroVisor uses a different Ethernet type for the network transmission. Then, those packets are passed to tools, such as rrdtool and rrdcached, to be aggregated and/or stored. Any aggregator that supports the RRD form can process these statistics.

The following type of metrics are currently provided by the monitoring API, captured by the RRD aggregator server and can be queried between time periods:

1. For every MicroVisor (i.e. at MicroVisor level)
  - 1.1. Timestamp
  - 1.2. Number of Physical CPUs (pCPUs)
  - 1.3. Number of CPU Pools
  - 1.4. Number of NUMA Nodes
  - 1.5. Number of Guests (VM domains)
  - 1.6. Total Memory
  - 1.7. Free Memory
  - 1.8. Average Load
2. Per Physical CPU (local)
  - 2.1. CPU ID
  - 2.2. CPU Time
  - 2.3. Timestamp
3. Per NUMA node (local)
  - 3.1. Node ID
  - 3.2. Total memory
  - 3.3. Free memory
4. Per CPU Pool (local)
  - 4.1. Cpu Pool ID
  - 4.2. Average Load
  - 4.3. Number of CPUs
  - 4.4. Number of Guest domains
5. Per Guest Domain (i.e. local VM level)
  - 5.1. VM domain UUID
6. Per vCPU (in domain)
  - 6.1. Cpu ID
  - 6.2. Timestamp
  - 6.3. Utilization percentage
7. Memory (in domain)
  - 7.1. Timestamp
  - 7.2. Current memory used
  - 7.3. Max memory available
8. Network VIFs (per Virtual Interface)
  - 8.1. VIF ID
  - 8.2. Timestamp
  - 8.3. Recv bytes

- 8.4. Recv packets
- 8.5. Recv errors
- 8.6. Recv drops
- 8.7. Transmit bytes
- 8.8. Transmit packets
- 8.9. Transmit errors
- 8.10. Transmit drops
- 9. Storage (per Virtual Block Device)
  - 9.1. VBD ID
  - 9.2. Timestamp
  - 9.3. Backing dev type
  - 9.4. Device name
  - 9.5. Outstanding IO requests (number)
  - 9.6. Read requests
  - 9.7. Write requests
  - 9.8. Read sectors
  - 9.9. Write sectors

The frequency of the generated messages can be configurable from a few seconds to minutes; however, very frequent updates will generate a lot of network traffic on large clusters. A frequency in the order of tens of seconds should be sufficient for most monitoring and orchestration uses.

### 2.6.2 Performance counters on Virtualization

Besides monitoring statistics for the platform, the integration with the ACTiManager and orchestration services requires more fine-grain and detailed statistics about the CPU behavior in VMs and applications running in VMs. For that purpose the MicroVisor implements performance counters at the hypervisor layer that provide a multitude of low-level, high-frequency metrics that can be used to characterize the behavior and performance bottlenecks of VMs and applications.

The performance counter layer and tools access the Performance Monitoring Unit (PMU) in the CPU cores, allowing a close look at the behavior of the hardware and its associated events, similar to the “perf” Linux tool. The MicroVisor allows monitoring a list of events to measure micro-architectural events in hardware, such as the number of cycles, instructions retired, LLC cache misses and so on. Those events are called PMU hardware events or hardware events for short, and they vary with each processor type and model<sup>5</sup>. Currently the MicroVisor provides support for these events on some common Intel x86 CPUs (e.g. Xeon / Skylake CPUs), while support for more hardware models will be gradually added.

Additionally the hypervisor itself is providing detailed metric for its own hypervisor software events, such as exceptions, vmexits, apic timer interrupts, interrupts, hypercalls, context switches, domain page TLB flushes, mmuext ops, calls to mmu\_update, page updates, and many more. Appendix I, Section 6.3 includes a detailed list of the current hypervisor performance

---

<sup>5</sup> Intel® 64 and IA-32 Architectures Software Developer’s Manual: Volume 3, Chapter 19, Table 19.1 <https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html>

counters used in order to measure the performance of each hypervisor node and VM domains running on it.

The current implementation includes a cmd-line tool named “mvctl-perfc” that can read the performance counters provided by the MicroVisor over the Ethernet and report them as output, as well as reset the counters if needed.

The mvctl-perfc tool currently has the following options:

- --perfc-dump (dumps all perfcounters)
- --perfc-get <perfcnter name> (dumps specific perfcounter)
- --perf-reset (resets perfcounters)

## 2.7 Performance

### 2.7.1 vNUMA Support

On Non-Uniform Memory Access (NUMA) architectures, memory accessing times of an application running on a CPU core depends on the relative distance between that specific CPU core and the specific memory. Modern server CPUs (e.g. Intel Xeons) are NUMA-based systems, where each CPU core has its own “local” memory, which they can access very fast, with low latency and high throughput. On the other hand, loading or storing data from and to remote memories (i.e. memories local to some other CPU cores in the system) is quite more complex and slow. NUMA machines are becoming more and more common, as the number of CPU cores increases.

NUMA awareness, that is the knowledge of the distance from each CPU to each memory node, has a significant performance impact on large machines, as soon as many VMs start running memory-intensive workloads on a shared host. In fact, the cost of accessing remote (non node-local) memory locations is high, and the performance degradation is likely to be noticeable. Published performance results on the Xen hypervisor for a memory-intensive benchmark with several competing VMs running concurrently on a hypervisor with 2 NUMA nodes, indicate up to 25% better performance when using all local memory vs. remote memories<sup>6</sup>.

vNUMA (virtual NUMA) support allows NUMA-awareness for a virtual machine for many virtual CPU cores (vCPUs), allowing the OS scheduler and applications in the VM to make NUMA-aware decisions on memory allocation and CPU core affinity. A vNUMA topology is currently defined as a set of parameters such as:

- number of vNUMA nodes
- distance table
- vnodes memory sizes
- vcpus to vnodes mapping
- vnode to pnode map (for NUMA machines).

In the MicroVisor implementation, the vNUMA topology is exposed to HVM guests to improve performance when running workloads on NUMA machines. vNUMA-enabled guests may be running on non-NUMA machines and thus having virtual NUMA topology visible to guests. In the

---

<sup>6</sup> Xen on NUMA Machines [https://wiki.xen.org/wiki/Xen\\_on\\_NUMA\\_Machines](https://wiki.xen.org/wiki/Xen_on_NUMA_Machines)

current MicroVisor implementation of vNUMA, the default behavior when creating an HVM instance is to split the memory among all available NUMA nodes in the system.

Currently, the MicroVisor implementation of vNUMA is being optimized and tested for regressions, in order to be merged into the main MicroVisor code. The MicroVisor without vNUMA support shows the following console output for the “show numa” command for one VM domain:

```
(NEX) [2019-07-16 09:06:58] Domain 6 (total: 8380935):
(NEX) [2019-07-16 09:06:58]      Node 0: 8380667
(NEX) [2019-07-16 09:06:58]      Node 1: 268
```

With NUMA support and the current vNUMA implementation, the MicroVisor console command output for memory allocated for a VM domain is the following:

```
(NEX) [2019-07-16 09:10:11] Domain 5 (total: 8384496):
(NEX) [2019-07-16 09:10:11]      Node 0: 4192248
(NEX) [2019-07-16 09:10:11]      Node 1: 4192248
(NEX) [2019-07-16 09:10:11]      2 vnodes, 32 vcpus, guest physical layout:
(NEX) [2019-07-16 09:10:11]      0: pnode   0, vcpus 0-15
(NEX) [2019-07-16 09:10:11]      0000000000000000 - 00000003ffffffff
(NEX) [2019-07-16 09:10:11]      1: pnode   1, vcpus 16-31
(NEX) [2019-07-16 09:10:11]      0000000400000000 - 000000080ffffffff
```

#### 2.7.1.1 Guest vNUMA support

On the current MicroVisor implementation of vNUMA, the NUMA information output on a vNUMA-enabled guest VM is presented below:

```
root@instance-4:/home/ubuntu# numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
node 0 size: 15771 MB
node 0 free: 15353 MB
node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
node 1 size: 16360 MB
node 1 free: 15912 MB
node distances:
node   0   1
  0:  10  20
  1:  20  10
root@instance-4:/home/ubuntu# numactl -s
policy: default
preferred node: current
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31
cpubind: 0 1
nodebind: 0 1
membind: 0 1
```

In contrast, a vNUMA-disabled guest VM on the MicroVisor shows no NUMA awareness, and all CPU cores appear to be on the same node:

```
root@instance-5:/home/ubuntu# numactl -H
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31
node 0 size: 32132 MB
node 0 free: 31463 MB
node distances:
node    0
  0:  10
root@instance-5:/home/ubuntu# numactl -s
policy: default
preferred node: current
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31
cpubind: 0
nodebind: 0
membind: 0
```

### 2.7.1.2 vNUMA Performance Impact

To demonstrate the impact of vNUMA and NUMA awareness in VMs on the MicroVisor, the Stream memory bandwidth benchmark<sup>7</sup> has been used to measure memory performance. For the purposes of the benchmark we created two HVM instances, each with 16 CPU cores, 32GB of memory and 100GB disk. One VM had vNUMA support enabled and the other vNUMA disabled.

The results shown below demonstrate clearly the performance advantages of NUMA-awareness with vNUMA. We observe that NUMA-enabled VMs with CPU pinning perform up to 228% better than non-NUMA-enabled VMs, also with CPU pinning enabled.

#### 1. NUMA-enabled guest VM (no CPU pinning):

<sup>7</sup> STREAM: Sustainable Memory Bandwidth in High Performance Computers  
<https://www.cs.virginia.edu/stream/>

```

$ ./stream-old
[snipped]
-----
Function      Rate (MB/s)  Avg time  Min time  Max time
Copy:         63710.9499    0.0017    0.0016    0.0065
Scale:        60024.5188    0.0018    0.0017    0.0066
Add:          66281.3917    0.0025    0.0023    0.0077
Triad:        66170.5880    0.0025    0.0023    0.0498
-----
Solution Validates
-----

```

## 2. NUMA-enabled guest VM (split cores, with manual CPU pinning):

```

$ ./stream-old
[snipped]
-----
Function      Rate (MB/s)  Avg time  Min time  Max time
Copy:         82174.4774    0.0013    0.0012    0.0058
Scale:        74124.3794    0.0015    0.0014    0.0064
Add:          84262.7325    0.0019    0.0018    0.0057
Triad:        80492.9712    0.0020    0.0019    0.0192
-----
Solution Validates
-----

```

## 3. NUMA-disabled guest (CPU pinning enabled by default in the resource group):

```

$ ./stream-old
[snipped]
-----
Function      Rate (MB/s)  Avg time  Min time  Max time
Copy:         34795.8188    0.0030    0.0029    0.0054
Scale:        33712.2696    0.0031    0.0030    0.0051
Add:          36936.8189    0.0041    0.0041    0.0044
Triad:        37256.2326    0.0041    0.0041    0.0060
-----
Solution Validates
-----

```

The following graph illustrates the memory throughput difference with and without vNUMA support.

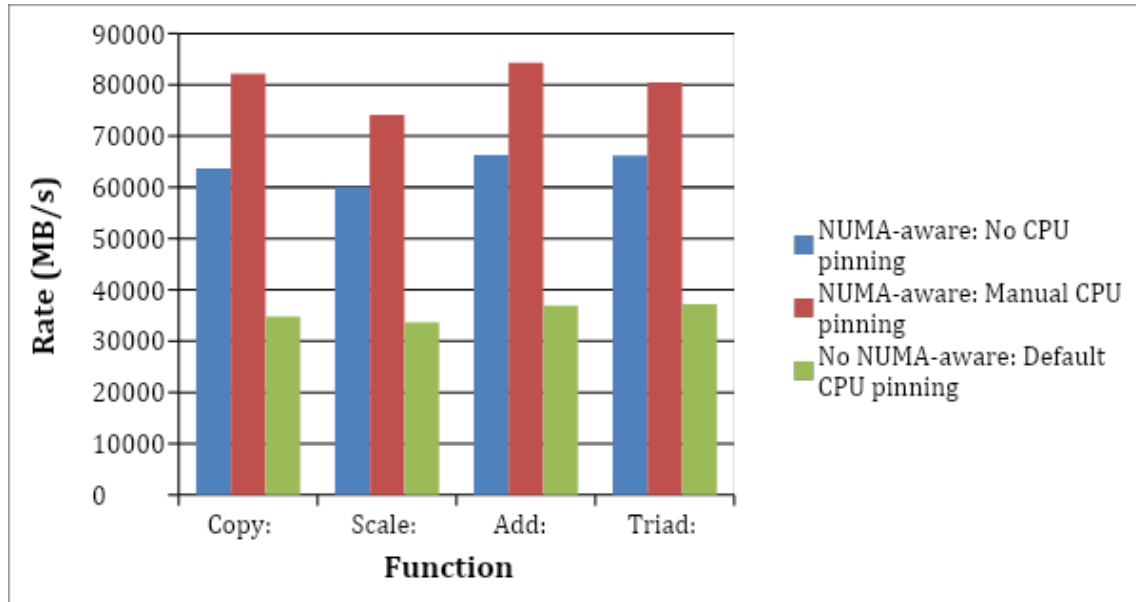


Figure 1: Memory throughput

## 2.8 Web-based User interface

In ACTiCLOUD a MicroVisor web-based User Interface (UI) component has been developed for configuring and controlling the platform and its resources on top of the control stack and services. The web UI has been designed and implemented by OnApp as a dynamic web front-end using the React javascript framework, running on the user's web browser, while the backend is provided by the MicroVisor management REST API implemented in a Go server, which interfaces with the control stack services to manage the MicroVisor platform resources.

Several components of the UI have been redesigned and improved in the Final MicroVisor prototype, managing racks, Hypervisors, networks, storage and instances (VMs). The Figures below display general views of the MicroVisor dashboard:



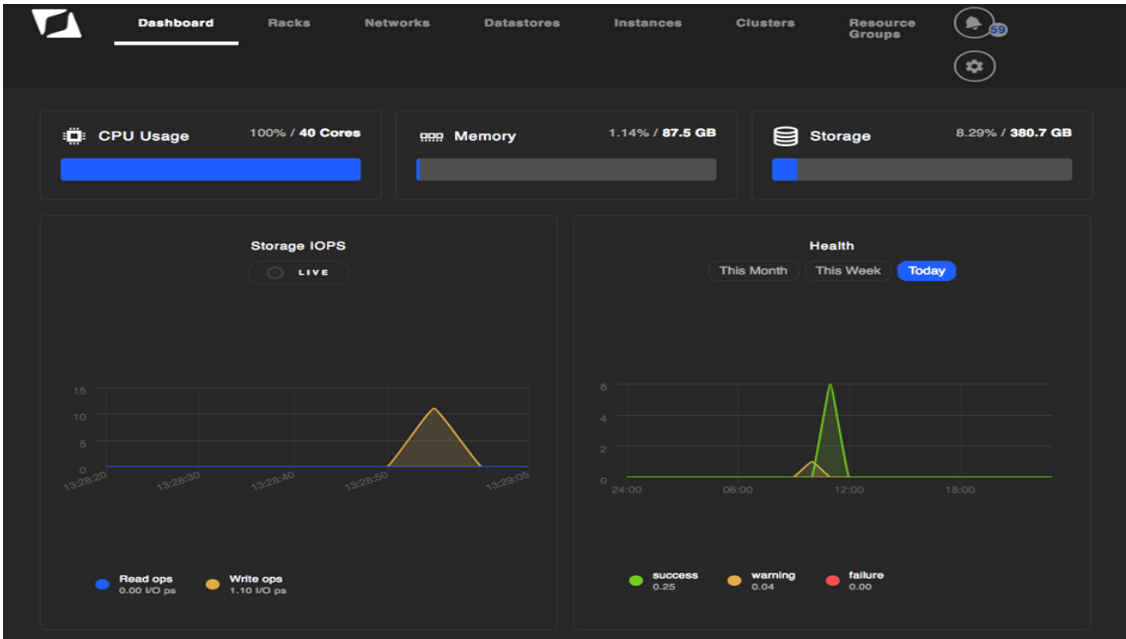


Figure 2: MicroVisor UI – Dashboard showing platform resources

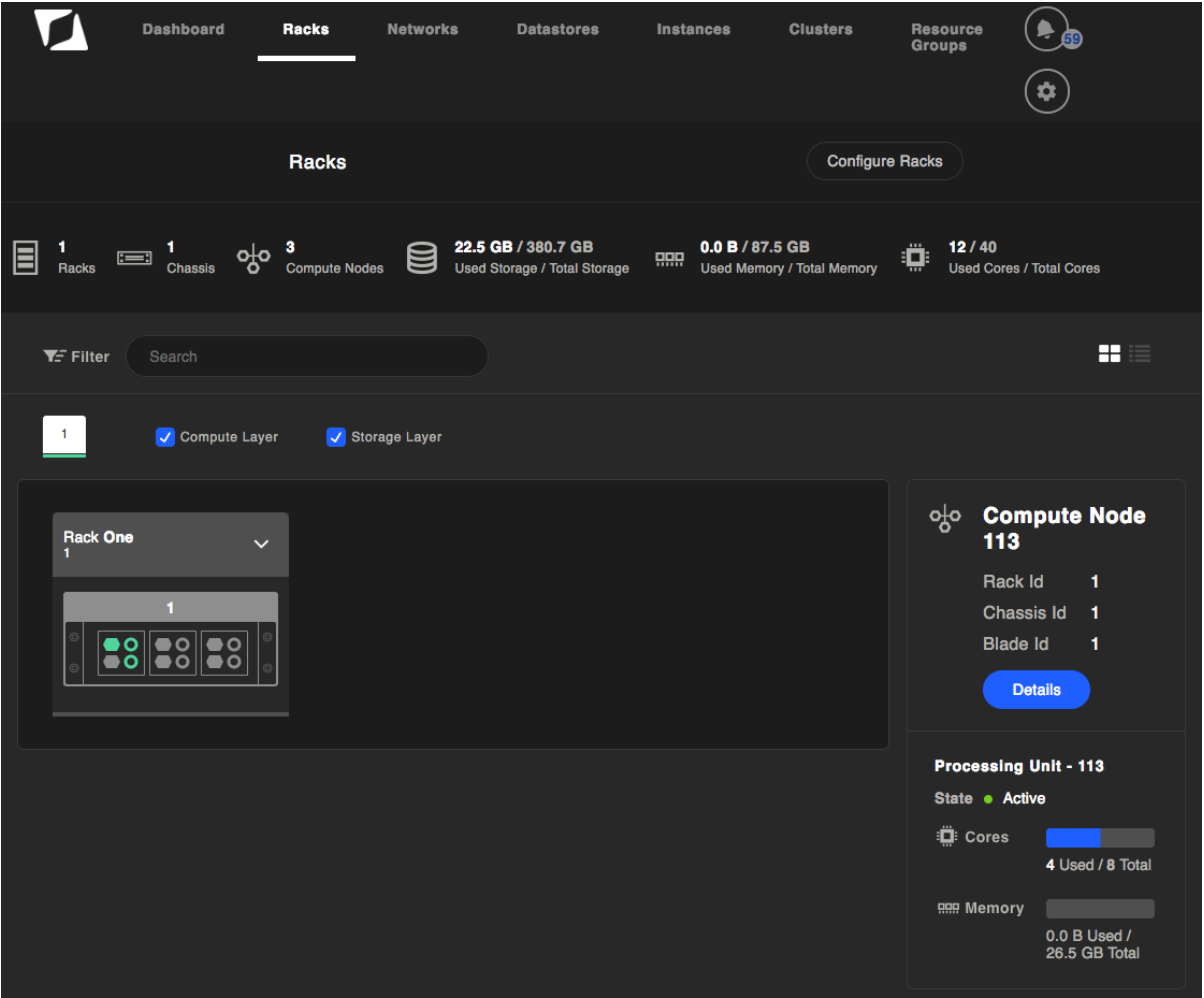


Figure 3: MicroVisor UI – Rack view showing their resources

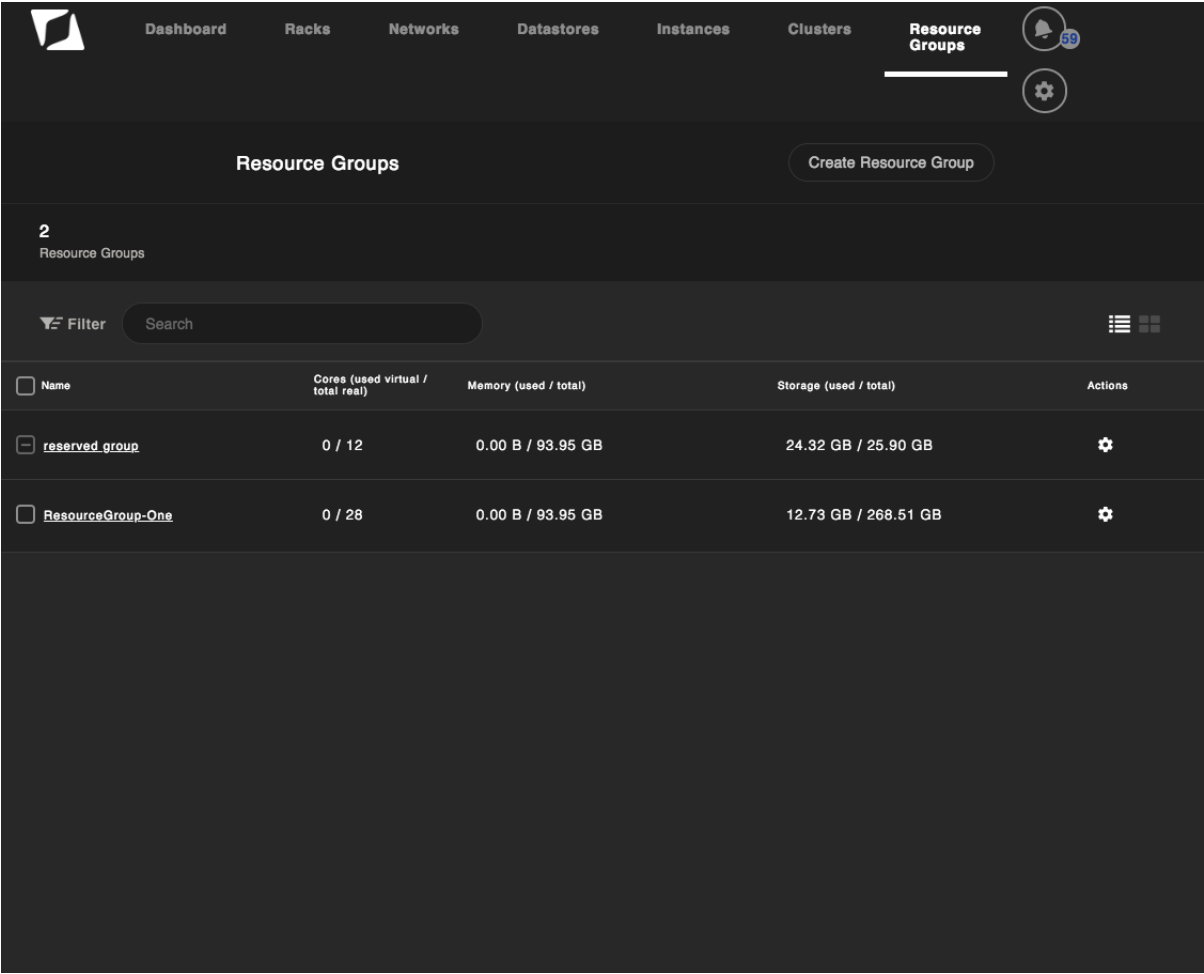


Figure 4: MicroVisor UI – Resource group view showing grouped resources

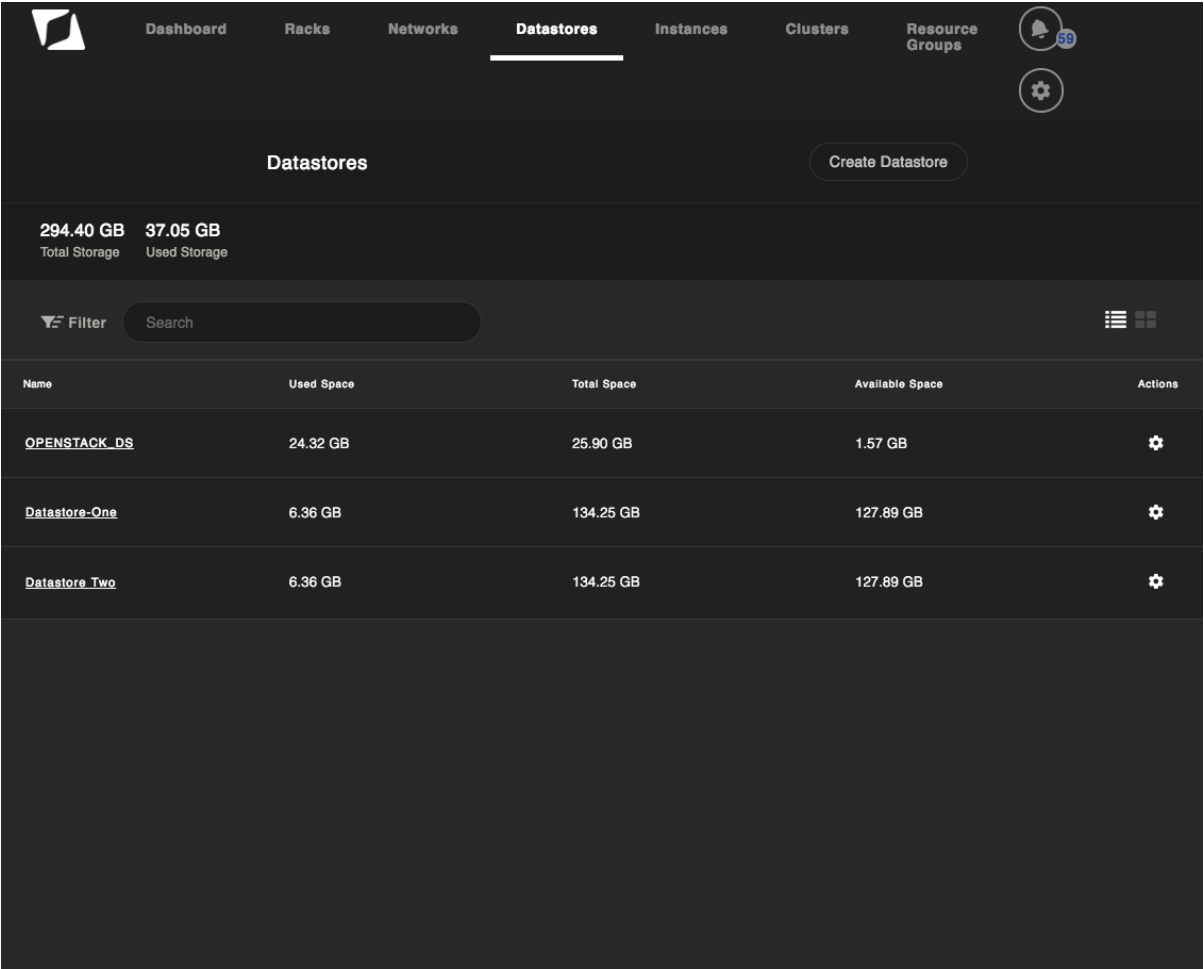


Figure 5: MicroVisor UI – Datastore view showing configured datastores and used storage capacity

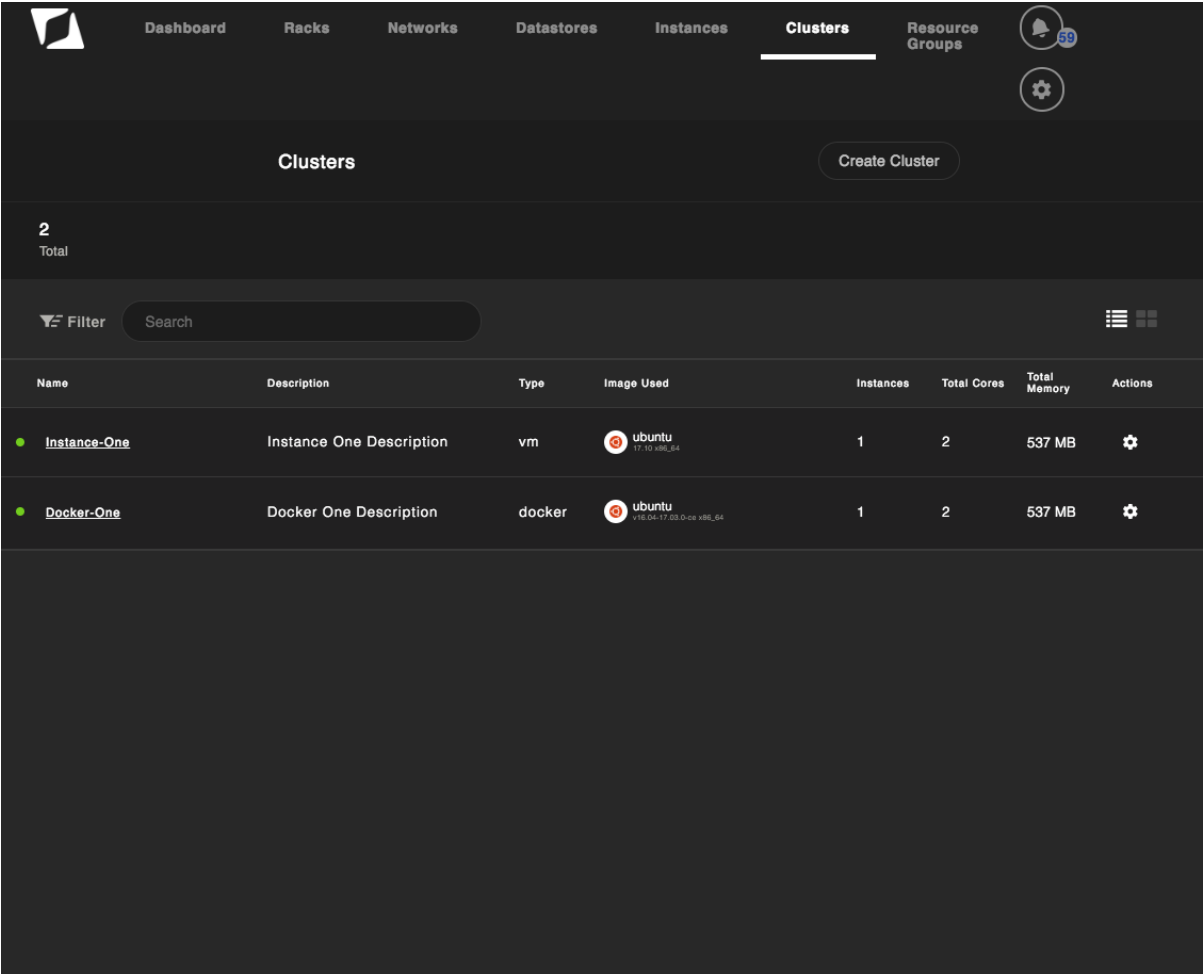


Figure 6: MicroVisor UI – Instance view showing virtual machines configured and deployed

### 3 Resource control in the MicroVisor platform

As mentioned in Section 2, in the MicroVisor platform a *resource group* is a combination of CPU cores, memory units and storage units that operates as a separate resource pool with common characteristics, such as storage performance, locality, CPU locality and pinning, overcommit, etc. These processing, networking and storage resources can be located anywhere in the cluster infrastructure (e.g. in one or more racks).

The notion of resource groups is essential to the MicroVisor and all virtual machines (VMs or server instances) must belong to a resource group, which is required upon VM creation time. The resource group for the VM identifies the set of physical CPU cores that the VM can be executed on (or pinned on if CPU pinning policy is selected), as well as the associated storage datastores and networks that the VM can use (e.g. a pool of fast NVMe flash storage, or a slower hard disk pool).

#### 3.1 Relevance to ACTiCLOUD Objectives

The resource groups are used to pool resources of VMs into separate isolated groups, in order to place VMs in different resource pools. The concept of resource groups can be used to provide:

- Tiered levels of service for VMs, such as a high-performance tier, using for example quick storage or network with pinned CPUs, or a slower tier, using shared overcommitted CPUs without pinning and slower storage.
- Guaranteed VM performance (e.g. VMs pinned on specific CPUs)
- Performance isolation for different VMs that can be noisy and affect others.
- Multiple reliability and availability levels of service for VMs.

This is directly related to the ACTiCLOUD Strategic Objective 1 (SO1): Effective utilization of cloud resources. This strategic objective is sought by unchaining resource management and provisioning from the physical bounds of a single server and a single cloud site, together with the implementation of novel resource-aware allocation policies. SO1 is further split into the following two explicit sub-objectives that are concurrently pursued:

SO1.1: Resource efficiency: Focusing on the requirements of Cloud Service Providers (CSPs) for reduced Total Cost of Ownership (TCO), ACTiCLOUD aims to drastically increase the resource efficiency of cloud infrastructures in terms of throughput per resource unit.

SO1.2: Performance stability: Focusing on the requirements of end-users, ACTiCLOUD aims to deliver performance stability to applications in terms of minimized performance variation compared to standalone execution.

#### 3.2 Performance features of Resource Groups

Several performance-related features in the MicroVisor platform are supported through resource group attributes, which are applied on all VMs configured in each resource group. These features are explained in more detail in this section.

##### 3.2.1 NUMA CPU assignment

When creating or editing a resource group, a system administrator controls which physical CPU cores are assigned to the group and therefore on which physical cores a VM in the group can execute on. The platform (i.e. the API, CLI and UI) provide NUMA hardware information to the

user configuring the resource groups. Also the web UI groups the CPU cores according to their physical NUMA nodes, so resource groups can be created using any combination of CPU cores within NUMA nodes or across multiple of them. This allows control of the NUMA physical CPU cores where VMs will be using, according to the user preferences for performance (i.e. exploiting the NUMA topology).

### 3.2.2 CPU overcommit

CPU overcommit on the MicroVisor allows more than one VMs to execute on the same physical CPU cores, which is useful in many multi-tenant workloads in VMs that do not require dedicated CPUs for performance. On the other hand, no CPU overcommit implies a 1-to-1 mapping of virtual CPU cores to physical cores and can provide increased CPU performance on dedicated physical CPUs. This resource group policy can be configured during resource group creation and applies to all VMs that belong to that group, allowing shared or dedicated CPU resources depending on VM needs.

### 3.2.3 CPU Pinning (Core Assignment)

CPU core assignment, or as it is widely known, *CPU pinning*, is performed through the Hypervisor and controlled via the management layer and the UI via resource groups. It essentially enables the binding and unbinding of a VM process to a CPU or a range of CPUs, so that the VM processes or threads will execute only on the designated CPU(s) rather than any random CPU the scheduler decides. CPU pinning takes advantage of the fact that remnants of a VM process that was run on a given processor may remain in that processor's state (for example, data in the cache memory) after another process was run on that processor. Scheduling that VM process to execute on the same processor improves its performance by reducing performance-degrading events such as cache misses.

CPU pinning is a property of a resource group, which means that every VM on a resource group created with the pinning enabled, will have CPU pinning on specific CPUs. When creating or editing a resource group, a system administrator controls which cores are assigned to the group and therefore on which physical cores a VM in the group can execute on.

### 3.2.4 Storage Optimisation

During resource group deployment the user can select the “Optimized Storage” option, which enables a policy to allocate one storage replica on the same physical server where a VM is running. This means that when we create a VM on a “Storage Optimized” resource group, one replica of all its virtual disks will be mapped to physical storage residing on the same server host.

Optimized local storage minimizes latency (and usually increases throughput) for all I/O read requests, which are sent only to the local replica, avoiding network transfers. Local storage performs particularly well with fast NVMe flash drives, which have lower latencies and higher throughput than common network interfaces and protocols (i.e. 10GBps NICs). Replication for I/O write requests, however, performs at network-bound speeds, since the data are replicated on storage drives located on a separate server, over the network.

## 3.3 Reliability and availability features of Resource Groups

Regarding reliability and availability in resource groups, the following capabilities are supported:

### 3.3.1 Fault-tolerance

The 'Fault Tolerance' option in a resource group allows specification of the Unit of fault tolerance required for VMs and storage (None/Blade/Chassis/Rack) and the maximum number of Units allowed to be lost while remaining in a fault-tolerant state. With this feature the configuration of a VM maintains redundancy across a blade, chassis or rack to satisfy fault-tolerance requirements for the resource group.

### 3.3.2 Failover

If the “Automatic failover” option of a resource group is enabled, a VM failure in that resource group would trigger automatic failover of the VM in another node that is configured in the same resource group (depends also on the fault-tolerance level selected). To support such a failover feature, the system requires a level of redundancy of resources, which is enforced by the resource group and VM configuration policies.

## 3.4 Configuration of Resource groups via the UI

Resource groups with CPU pinning and overcommit, as well as binding to specific storage and network resources are a core concept of the MicroVisor platform and it is essential for all resource configuration to use them for the system operation.

To demonstrate the concept and provide a clear view of the usage and implementation of MicroVisor resource groups, we present in this section their configuration through the MicroVisor web UI. The process has been simplified through a “wizard” which gathers all necessary information for the creation of a resource group, which maps to all layers of the system through the controller stack and API. We believe this demonstrates the concept and the control features that are essential to ACTiCLOUD objectives for efficient resource management.

In the MicroVisor platform all virtual machines (also noted as VMs or server instances) must belong to a resource group. The resource group is identified by a set of physical CPU cores on one or more server hosts that are members of the group and is associated with a number of storage datastores and networks.

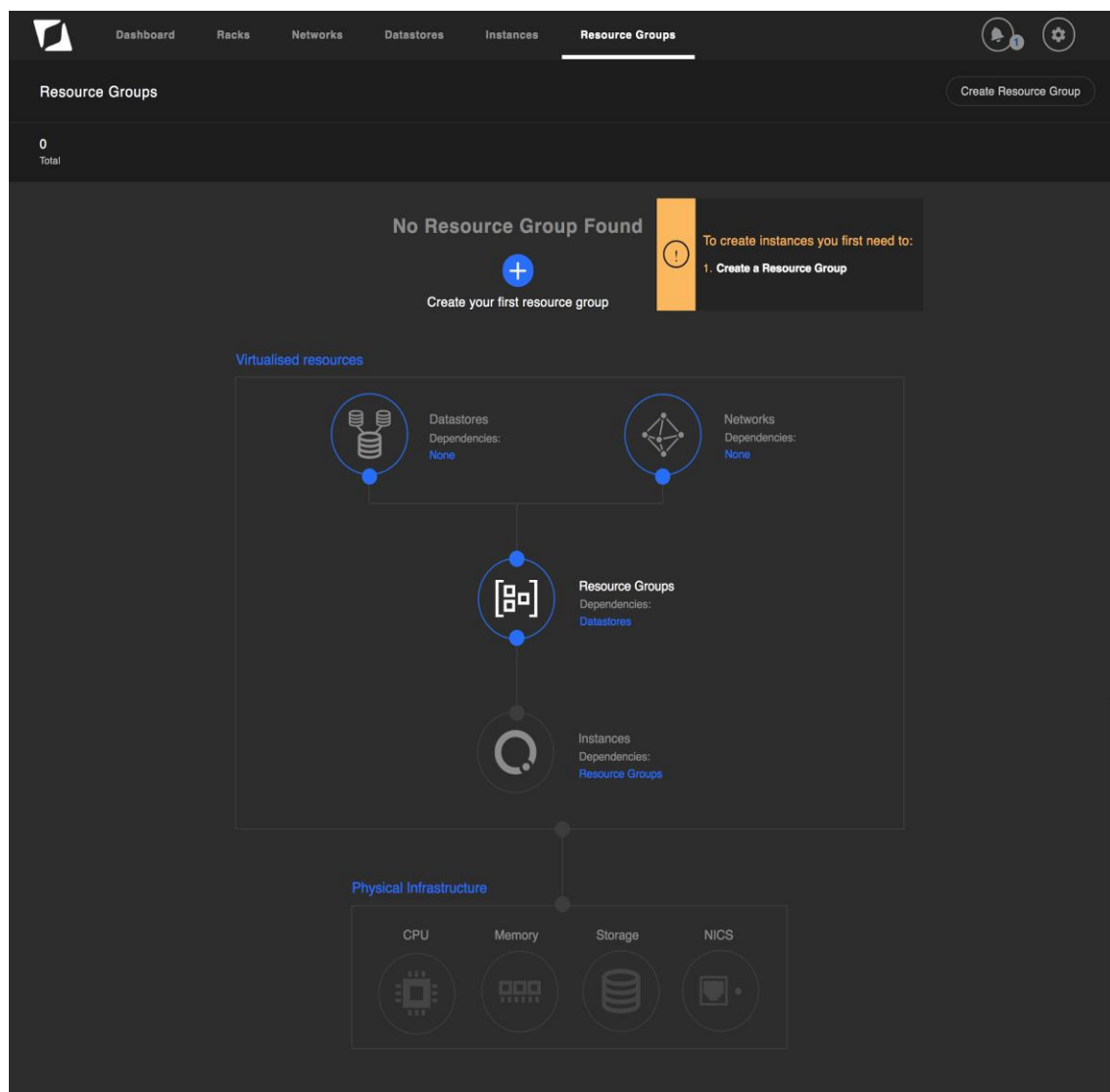
In order to create a resource group through the MicroVisor UI, the administrator must:

1. Select the Resource Group tab.
2. Click the Create Resource Group button.
3. Fill in the configuration form through the following steps.

Initial Setup - No Resource Group available:

In case of an install in a brand-new system (i.e. that has not been previously configured), the Resource Groups menu option in the UI should look like the following:





**Figure 7: Resource group configuration**

In order to properly create a Resource Group, the steps described below must be followed:

**Step 1 - Select Storage Optimisation / Enable Fault Tolerance**

The Storage Optimisation option is enabled by default.

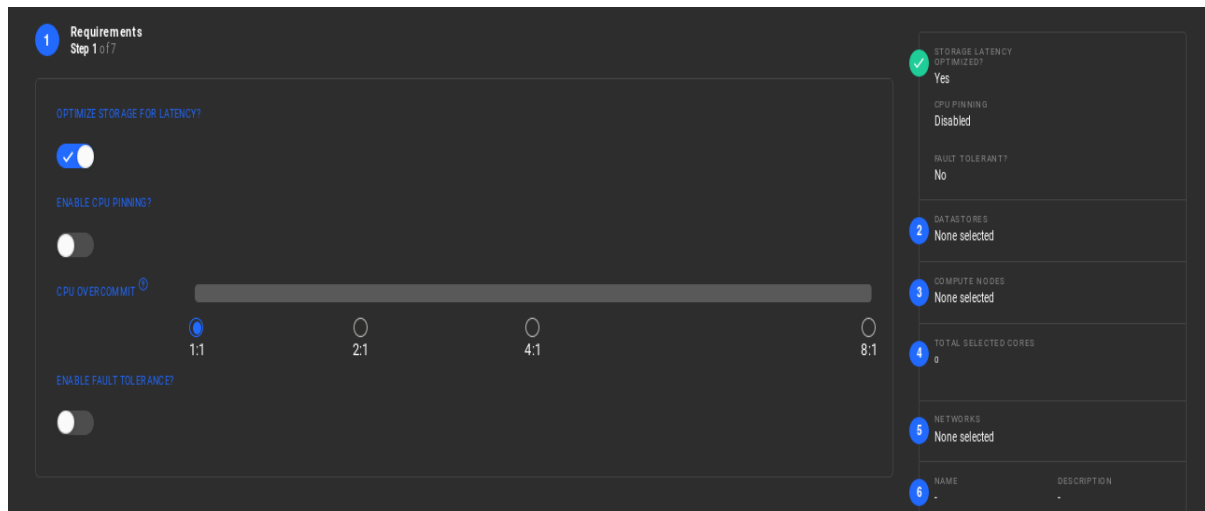


Figure 8: Storage optimization

In the context of cores assignment, 'CPU overcommit' attribute is supported and can be applied across all the selected cores. This feature provides the capability to allocate more than 1 virtual cores to a single physical core. Specifically, users can choose between the range of 2 to 8 virtual cores to be assigned to one single core.

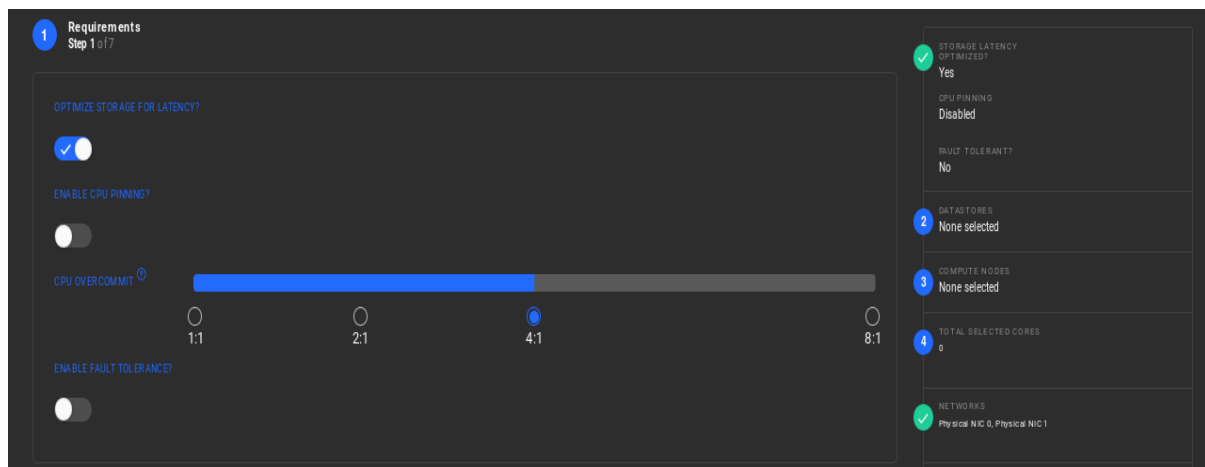


Figure 9: CPU overcommit

By enabling 'CPU Pinning' attribute, each virtual core corresponds to only one physical core, and as a result 'CPU overcommit' is disabled.

The 'Enable Fault Tolerance' option allows specification of the Unit of fault tolerance required (Blade/Chassis/Rack) and the maximum number of Units allowed to be lost while remaining in a fault tolerant state.

The option to Automatically Failover in the case of a failed Unit is also available.

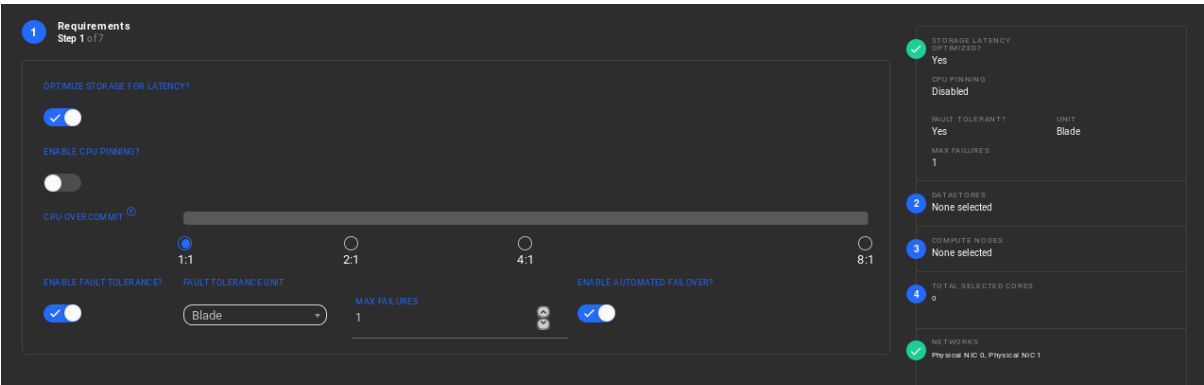


Figure 10: Fault tolerance

**Step 2 - Select available datastores that will hold VM data**

Select one or more data stores that are accessible to the resource group according to step 1 selections

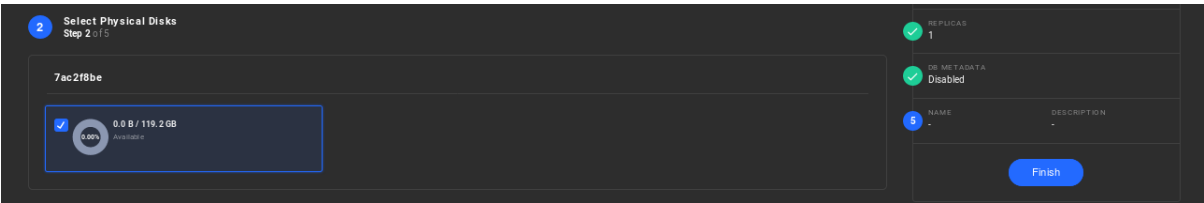


Figure 11: Datastores selection

**Step 3 - Select the compute node resources (MicroVisors)**

Select the members (compute nodes) that you want to make part of the resource group.

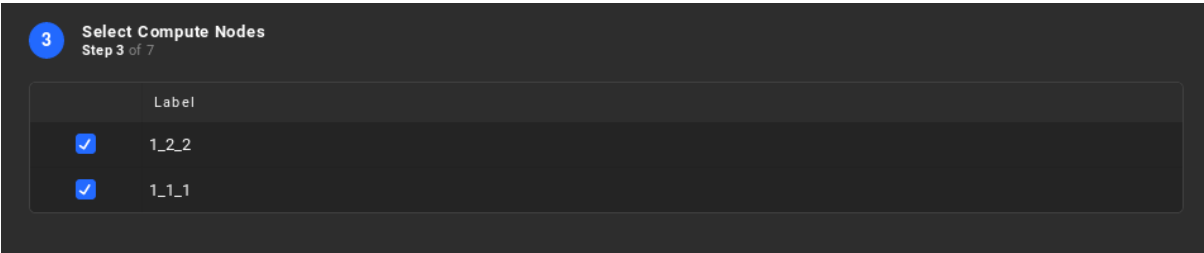


Figure 12: Nodes selection

**Step 4 - Select the cores that will be assigned to the group**

Select the cores on each member to assign to the resource group.

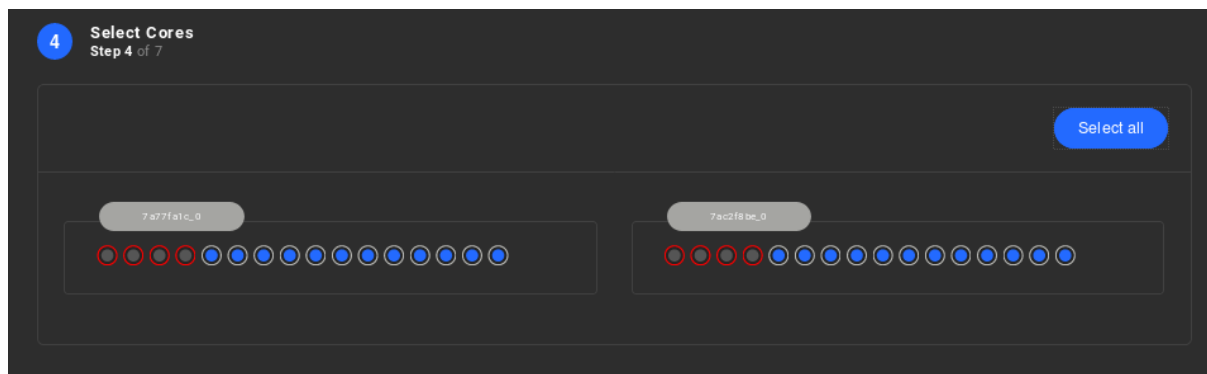


Figure 13: Cores selection

**Step 5** - Assign the networks to which the instances may have access

Select one or more networks that are accessible to the resource group.

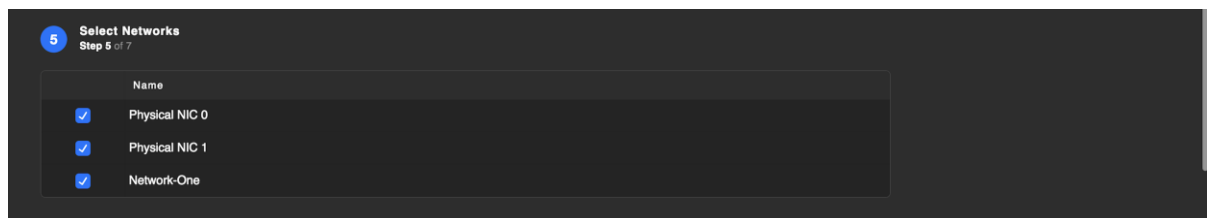


Figure 14: Networks selection

**Step 6** - Finalize the configuration

Enter a name and relevant description. Click the Finish button to complete the resource group creation.

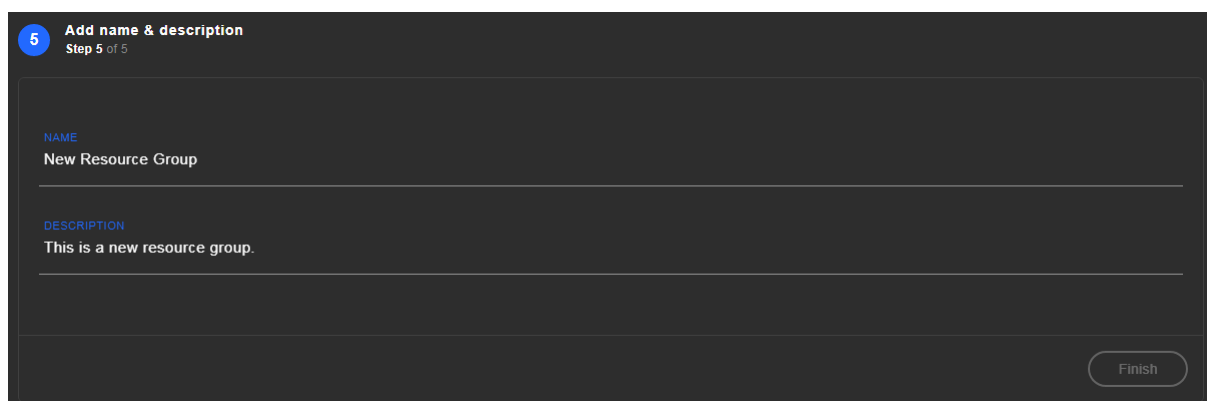


Figure 15: RG configuration finalization

## List of Resource Groups

In case the system has been previously configured, the Resource Groups menu option would display them in a list, as shown in the following example:

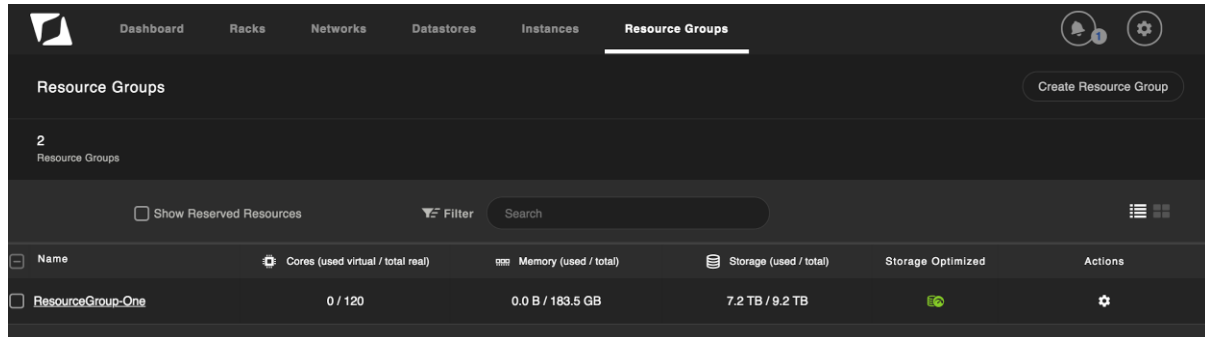


Figure 16: RG availability

### 3.4.1 Command-line support for resource groups

There are several commands through the CLI to manage resource groups, such as create, delete, edit resource groups, as well as add and remove network or datastores to them. For reasons of brevity we only present the CLI commands to create a resource group with their options. All the available commands can be found in our CLI documentation.

```
COMMAND:
    create_group
NAME:
    - Create a new resource group.

USAGE:
    [command options] [arguments...]

DESCRIPTION:
    Create a new resource group.
    Example:
    osd create_group --name=super --core_ids=54424488_4,54424488_5 --network_ids=66,2 --
    datastores=fjxi3vnzg4hm9y

OPTIONS:
    --json                                Return the output as json.
    --name value                          The display name of the object.
    --description value                   The description of the object.
    --core_ids value                      The list of physical cores, based on their IDs
    --network_ids value                  The list of networks, based on their IDs
    --datastores value                   The list of datastores, based on their IDs
    --cpu_overcommit value               The maximum number of extra vCPUs per pCPU for each
                                          VM. For example cpu_overcommit=2 will let each
                                          physical core run 2 virtual cores. (default: 0)
    --disable_storage_latency_optimization This attribute will be used when creating an
                                          instance to prioritize the MicroVisor that has
                                          at least one local physical disk from the selected
                                          datastore.
    --enable_cpu_pinning                 This attribute will enable the CPU pinning for all
                                          the instances created for this group.
    --filter value                       Filter the values you want to see from the output.
    --enable_ft                          Enable the fault tolerance.
    --ft_unit value                     Choose one of the fault tolerance units: 'mvgroup',
                                          'blade', 'chassis' and 'rack'
```

<code>--ft_max_failures value</code>	The number of MicroVisors that are acceptable to fail. (default: 0)
<code>--disable_ft_auto_failover</code>	Disable the fault tolerance's automation failover

### 3.4.2 API support for Resource groups (resourceGroupsPost)

There are several calls for managing resource groups in the MicroVisor API, as all the operations of the UI are handled via the REST API (e.g. create/delete/edit resource group). For reasons of brevity we only present the resourceGroupsPost API endpoint (`/resource_groups`), which creates a new resource group. More details and all the available REST API endpoints and their options can be found in our API documentation.

`/resource_groups`

On the following scenarios, the endpoint will fail with 422 validation error:

- The name cannot be empty. (422)
- No core ids are assigned. (422)
- No datastore ids are assigned. (422)
- No network ids are assigned. (422)
- Unknown core, datastore or network ids are given. (422)
- The `cpu_overcommit` value is less than 1. (422)
- The `cpu_overcommit` value is greater than 8. (422)
- The CPU pinning is enabled and the `cpu_overcommit` is over 1. (422)

#### 3.4.2.1 Usage and SDK Samples

Curl:

```
curl -X POST "http://localhost/api/resource_groups"
```

#### 3.4.2.2 Parameters

- Body parameters

Name	Description
resourceGroup	<pre>{   Required: cores,cpu_overcommit,datastores,name,network_ids   name: string   description: string   datastores: [     <i>The datastore ids that will be assigned to the resource group.</i>     String ]   network_ids: [     <i>The network ids that will be assigned to the resource group.</i>   ] }</pre>

	<pre>integer (int64) ] cores: [ <i>The core ids that will be assigned to the resource group.</i> String ] cpu_overcommit: integer (int32) minimum:1 <i>The number of extra virtual cores that can be created over a physical core. For example, a cpu_overcommit value of 2 means that 2 virtual cores can be created for a single physical core.</i> is_storage_latency_optimized: boolean <i>This attribute will be used when creating an instance to prioritize the MicroVisor that has at least one local physical disk from the selected datastore.</i> is_cpu_pinning_enabled: boolean <i>This attribute will enable the CPU pinning for each instance created in the resource group. CPU pinning means that the system will pin each virtual core of the instance to a physical core. The default is false.</i> fault_tolerance: {   is_enabled: boolean   unit: string   max_failures: integer   automated_failover: boolean   is_degraded: Boolean } }</pre>
--	---

### 3.4.2.3 Responses

#### 3.4.2.3.1 Status: 202 - Resource group successfully created.

```
{
```

*A resource group is a combination of CPU cores, memory units and storage units. These resources can be located anywhere in the infrastructure.*

Required: id

id: integer (int32)

name: string

description: string

datastores: [

*The ids of the datastores that are assigned to the resource group.*

String ]

disk\_ids: [

*This is an array containing all the disk ids that belong to all the datastores that are assigned to this resource group.*

String ]

network\_ids: [

*This is an array containing all the network ids that are assigned to this resource group.*

integer (int64) ]

cores: [

*The ids of those cores that are assigned to the resource group.*

String ]

virtual\_cores: integer (int32)

*The total number of virtual cores created by instances. These virtual cores are using the physical cores that are assigned to the resource group by definition.*

real\_cores: integer (int32)

*The total number of physical cores.*

core\_usage: number (float)

*The percentage of the physical cores of the resource group that are currently used by the virtual cores of the instances.*

cpu\_overcommit: integer (int32) minimum:1

*The number of the extra virtual cores that can be created over a physical core. For example cpu\_overcommit 2 means that the scheduler allocates up to 2 virtual cores per physical core.*

avail\_memory: integer (int32)

*The available memory from the processing units in the resource group. The value is in MB.*

total\_memory: integer (int32)

*The total memory from the processing units in the resource group. The value is in MB.*

total\_storage: integer (int32)

*The total space of the datastores in the resource group. The value is in MB.*

avail\_storage: integer (int32)

*The available space of the datastores in the resource group. The value is in MB.*

is\_reserved: boolean

*Denotes if the resource group is reserved which means that it is used by OpenStack and/or other services.*

is\_storage\_latency\_optimized: boolean

*This attribute will be used when creating an instance to prioritize the MicroVisor that has at least one local physical disk from the selected datastore.*

}



### 3.5 Storage & Configuration of Datastore

As it has already been mentioned in Section 2, a *Datastore* is a collection of physical drives with a redundancy and overcommit policy applied. All the data that is stored on the system is thin provisioned so the available physical storage can be “overcommitted” up to any amount. It has to be noted that any overcommitment of storage comes with an associated risk that the system may run out of physical storage if excessive data is stored on the disks. In order to present local storage drives to the server workloads as a unified storage layer, the available drives must be selected and a distributed “Datastore” should be created across them.

In order to configure Datastores, the steps described below must be followed:

1. Click the Datastores tab.
2. Click the Create Datastore button.

Fill in the storage configuration form step by step:

In case no Datastore has been previously configured - the "Datastores" menu option should look like the following screenshot:

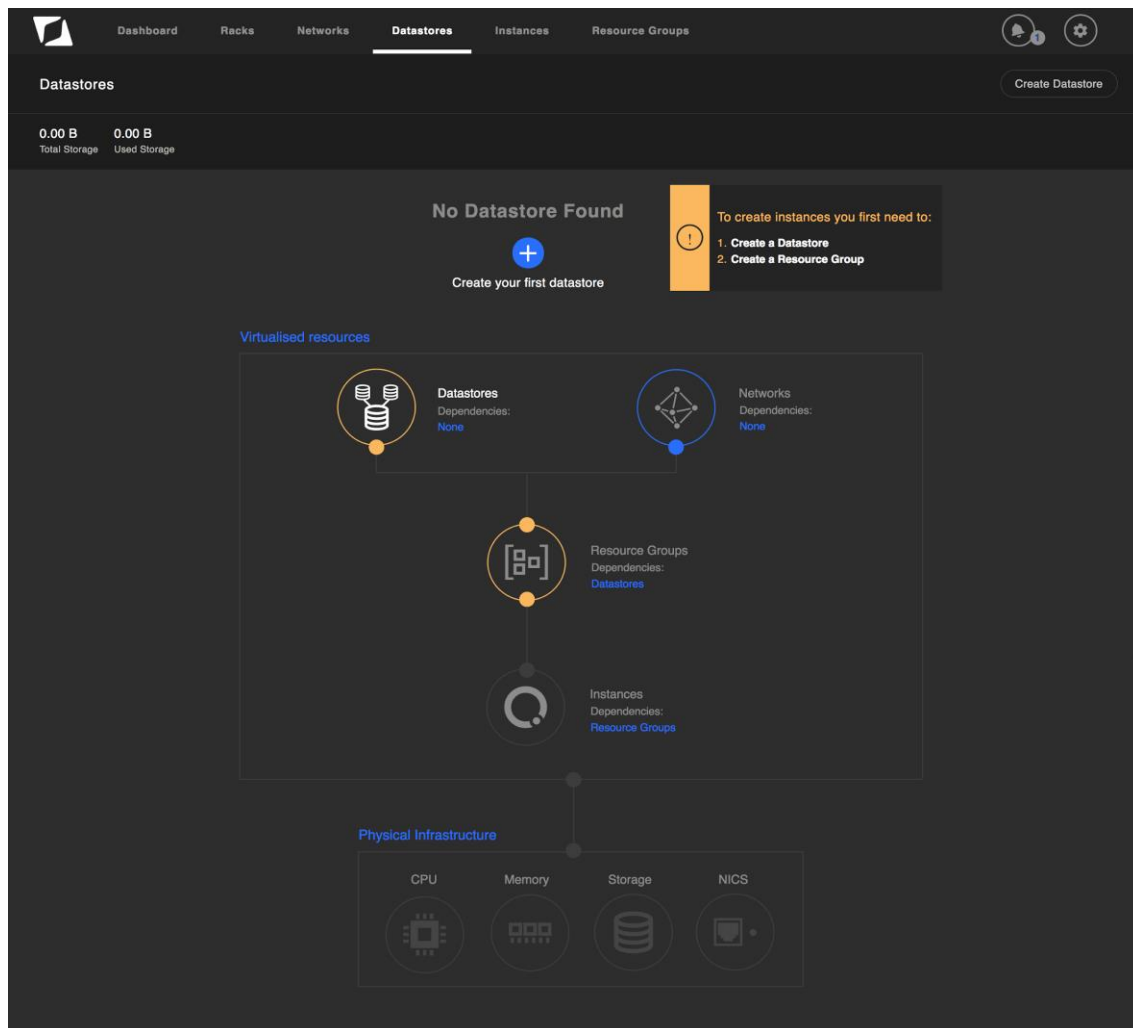


Figure 17: Creating a datastore

**Step 1 - Select the Compute Resources:**

Select the compute nodes with the physical storage to be added to the Datastore. This will populate the available storage drives that are present across all the selected compute nodes.

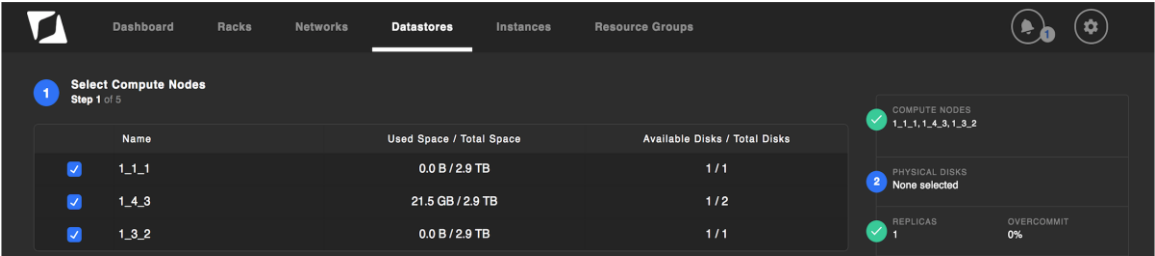


Figure 18: Compute nodes selection

**Step 2 - Select the drives:**

Select the drives to be added into the Datastore (multiple drives required if fault-tolerance with multiple replicas are needed).

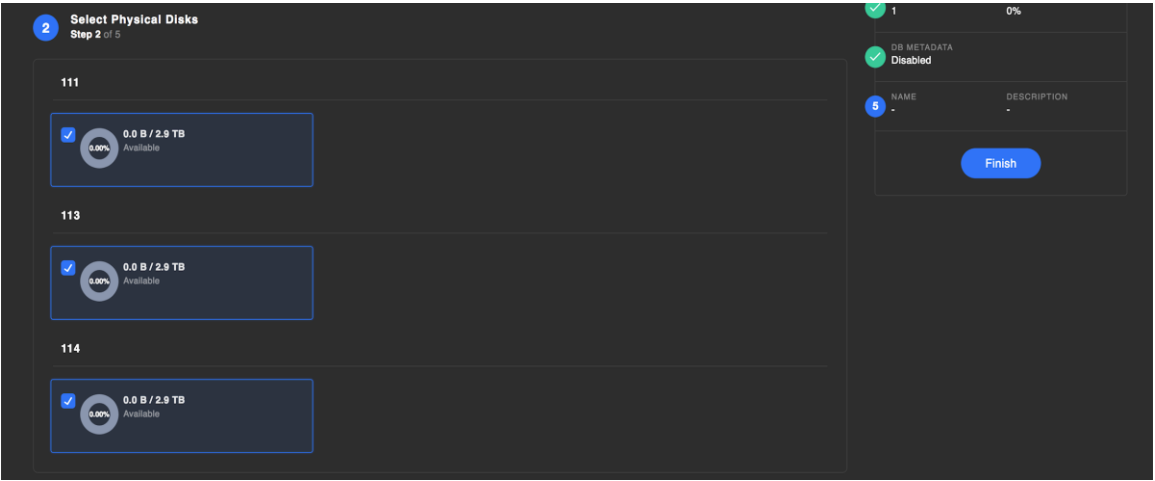


Figure 19: Physical disks selection

**Step 3 - Select the Redundancy:**

Select the redundancy level to be applied (1 or 2 replicas).

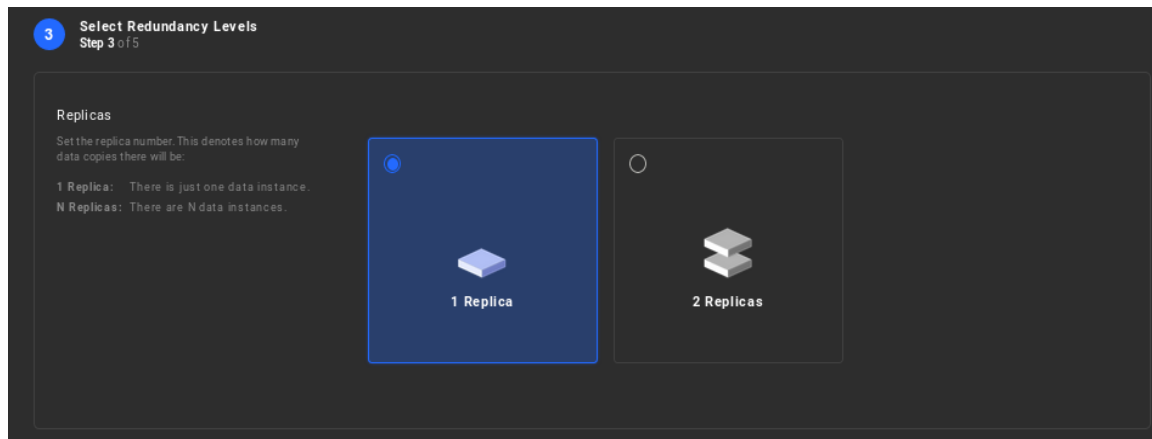


Figure 20: Datastores redundancy

**Step 4 - Enable DB metadata - (system DB backup):**

Enable metadata, in order for the platform to automatically back up all system databases and restore data from a specific backup (this is for the high-availability of the controller metadata, which can be replicated on multiple storage devices).

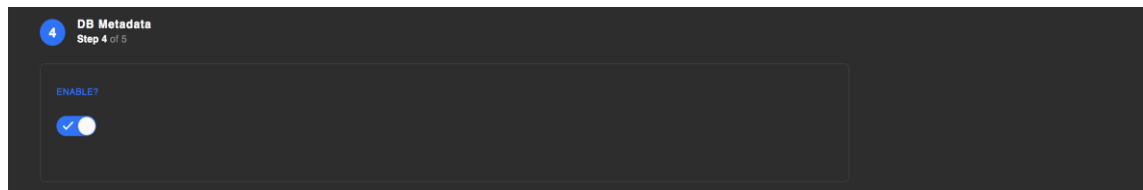


Figure 21: Enable metadata

**Step 5 - Finalizing the Datastore implementation:**

Assign a Name and a Description to the new Datastore that has been created. This information will be displayed in the Datastore list and will be used to identify the storage group when assigning to a resource group. Selecting "Finish" will create the Datastore.

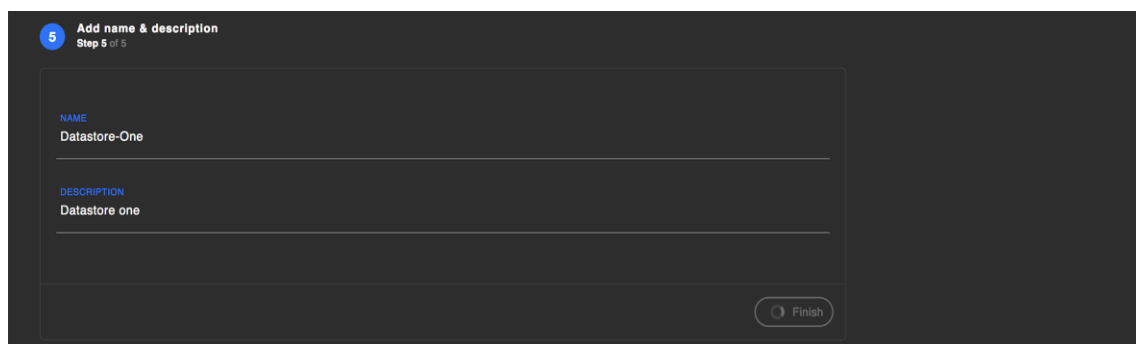


Figure 22: Finalizing Datastore

If the system has been configured properly, the Datastores menu option should look like the following example:

Name	Used Space	Total Space	Available Space	DB Metadata	Actions
<a href="#">Datastore-One</a>	1.1 TB	3.1 TB	2.0 TB	✓	
<a href="#">Datastore-Two</a>	1.1 TB	3.1 TB	2.0 TB	No	

Figure 23: Created Datastores list

### 3.5.1 Command-line support for datastores

There are several commands through the CLI to manage datastores, such as create, delete, edit datastores, as well as add and remove disks to datastores. For reasons of brevity we only present the CLI commands to create a datastore and list datastores with their options. All of the available commands can be found in our CLI documentation.

```
COMMAND:
    create_datastore
NAME:
    - Create a new datastore.

USAGE:
    [command options] [arguments...]

DESCRIPTION:
    Create a new datastore.
Example:
    osd create_datastore --name=test --disks=3584916917,3283879690 --replicas=1 --thick=1

OPTIONS:
    --json                Return the output as json.
    --name value          The display name of the object.
    --disks value         The list of storage disk's IDs.
    --overcommit value    The overcommit value of the space that the volumes can take.
    --replicas value      The number of replicas we want for the volumes. (default: 0)
    --thick value         The attribute to make the datastore thick. (default: 0)
    --db metadata         Enable a vdisk for backing up the DBs.
    --filter value        Filter the values you want to see from the output.
    --trigger_repair value How to trigger the repairs when the volumes are degraded,
                        'periodic' or 'manual'.
    --periodic_repair_sec value When the trigger_repair is periodic, then we can write in
                        seconds the periods it will check the volumes. (default: 0)
    --num_of_repairs value The number of repairs it can do in a parallel. (default: 0)

##
```

COMMAND:

```

get_datastores
NAME:
  - Get the list of datastores.

USAGE:
  [command options] [arguments...]

DESCRIPTION:
  Get the list of datastores.
  Example:
  osd get_datastores

OPTIONS:
  --json          Return the output as json.
  --filter value  Filter the values you want to see from the output.
  --if-not value  Don't show objects if the value is false.
  --if value      Don't show objects if the value is true.

```

### 3.5.2 API support for Datastores (datastoresPost)

There are several calls for managing datastores in the MicroVisor API, as all the datastore operations of the UI are handled via the REST API (e.g. create/delete/edit datastores). For reasons of brevity we only present the datastoresPost API endpoint, which creates a new datastore. More details and all the available REST API commands can be found in our API documentation.

### 3.5.2.1 API endpoint - `datastoresPost`

This API endpoint creates a new datastore.

The following scenarios will fail:

- The name is empty. (422)
- The name has to be less than or equal to 31 characters (422)
- The name cannot be 'OPENSTACK\_DS' (422)
- The name cannot have any of these characters '=', '?', '\', '\'', '\"' (422)
- The replicas is 0 or lower. (422)
- The overcommit is over 100. (422)
- The array of disks is empty or it contains non-existed IDs (422)

```
/datastores
```

### 3.5.2.2 Usage and SDK Samples

```
curl -X POST "http://localhost/api/datastores"
```

### 3.5.2.3 Parameters

## Body parameters

Name	Description
datastore	{ Required: disks,name,replicas name: string

	<p><i>The name of the datastore.</i></p> <p>disks:[ <i>The physical disks in IDs that the datastore will include. This can not be an empty array.</i></p> <p>String ]</p> <p>overcommit_disk: string</p> <p><i>Overcommit disk is in how much percent we can allow a vdisk to take more space than it really has.</i></p> <p>Enum: 0, 20, 50, unlimited</p> <p>replicas: integer (int64)</p> <p><i>This represents the number of the replicas we want for the stored data. It needs to be at least 1.</i></p> <p>repair_volume_policy: { Defines the policy on how to repair vdisks.}</p> <p>all of: { }</p> <p>has_db_metadata: boolean</p> <p><i>Defines if the datastore has a LUN for DB Metadata.</i></p> <p>is_thin: boolean</p> <p><i>Defines if the datastore has thin or thick provisioning.</i></p> <p>}</p>
--	--

#### 3.5.2.4 Responses

##### 3.5.2.4.1 Status: 202 - Datastore successfully created.

```
{
  Required: id
  id: string (uuid)
  name: string
  Defines the name of the datastore.
  replicas: integer (int64)
  Defines the number of replicas of the datastore.
  overcommit_disk: string
  Defines in percentage, on how much more space can a vdisk take.
  Enum: 0, 20, 50, unlimited
  reserved: boolean
  Defines if the datastore is reserved for the OpenStack's vdisk.
  total: integer (int64)
  The total space of the datastore.
```

used: integer (int64)

*The used space of the datastore.*

orphan\_vdisks: integer (int64)

*The orphan vdisks number of the datastore.*

available: integer (int64)

*The available space of the datastore.*

disks: [

*The objects of the physical disks that the datastore uses.*

{

Required: id

id: string

mvgroup\_id: string

*The MicroVisor group where the disk is located.*

used: number (float)

*The used percentage of the disk.*

total: integer (int64)

*The total size in bytes.*

vendor: string

datastore: string

*The datastore's ID*

is\_up: boolean

*If it is true then the disk is enabled*

state: string

*The current state of the disk*

Enum: enabled, disabled, error

serial\_number: string

*The serial number of the disk.*

ip: string

*The IP of the storage node that contains the disk. The storage node is a VM in the MicroVisor that contains some physical disks and a server called onappstore API. The onappstore API provides, to the OpenStack, the utility to use the physical disks. The IP can be connected only from the OpenStack.*

}

]

resource\_group\_ids: [

*The resource group's IDs that use the datastore.*

integer (int64) ]

state: string

*The state of the datastore.*

Enum: enabled, disabled

repair\_volume\_policy: {

*Defines the policy on how to repair vdisks.*

all of:

{ trigger: string

period\_seconds: integer

}

*{ It is a policy for repairing volumes The default attributes for this policy are 'trigger':manual 'num\_of\_repairs':null However the 'trigger' can not be 'config' or 'cleanup', because it does not make sense to repair on the creation or deletion of the volume*

num\_of\_repairs: integer

}

has\_db\_metadata: boolean

*Defines if the datastore has a LUN for DB Metadata.*

is\_thin: boolean

*Defines if the datastore has thin or thick provisioning.*

}

#### 3.5.2.4.2 Status: 422 - One or more validations have failed during a request processing.

Response Example

{}

all of:

{ Required: code,message

code: integer (int32)

*This is represents an error code. While the message of the error may change, the error code will stay the same. All these codes will be included in the documentation with a more descriptive text.*

message: string

*A human-readable message, describing the error.*

}

{

errors: [

{}

all of:

{ Required: code,message



code: integer (int32)

*This is represents an error code. While the message of the error may change, the error code will stay the same. All these codes will be included in the documentation with a more descriptive text.*

message: string

*A human-readable message, describing the error.*

}

{ Required: field

field: string

*The field name or id that this error is related to.*

detail: string

*It is a more detailed error message. }*

]

}

#### 3.5.2.4.3 Status: 500 - An unexpected error occurred.

##### Response Example

---

{

Required: code,message

code: integer (int32)

*This is represents an error code. While the message of the error may change, the error code will stay the same. All these codes will be included in the documentation with a more descriptive text.*

message: string

*A human-readable message, describing the error.*

}

## 4 Integration of the MicroVisor into OpenStack

The role of MicroVisor’s cloud management layer and the integration of the OpenStack cloud management software stack with the MicroVisor virtualization platform are analyzed in this section, within the context of the Final prototype.

### 4.1 Managing rack-scale resources through OpenStack

The MicroVisor cloud manager layer controls and orchestrates large pools of compute, storage, and networking resources in a rack-scale or larger-scale cloud. It provides mechanisms to monitor, deploy, migrate, and terminate hosted applications. While large industrial players in cloud resource provisioning, such as Amazon, Microsoft, Google, and VMware, rely on in-house commercial software to manage their own clouds, OpenStack has become the defacto choice of most other public cloud providers and private in house clouds.

ACTiCLOUD has embraced OpenStack as the cloud manager for the ACTiCLOUD platform, due to its open standards within the industry and its level of adoption in both small and large scale deployments. ACTiCLOUD aims to design and implement a hierarchy of resource schedulers that will operate dynamically at the rack and site levels by extending existing policies and collaborating with OpenStack, in order to maximize our impact and promote cloud interoperability and openness.

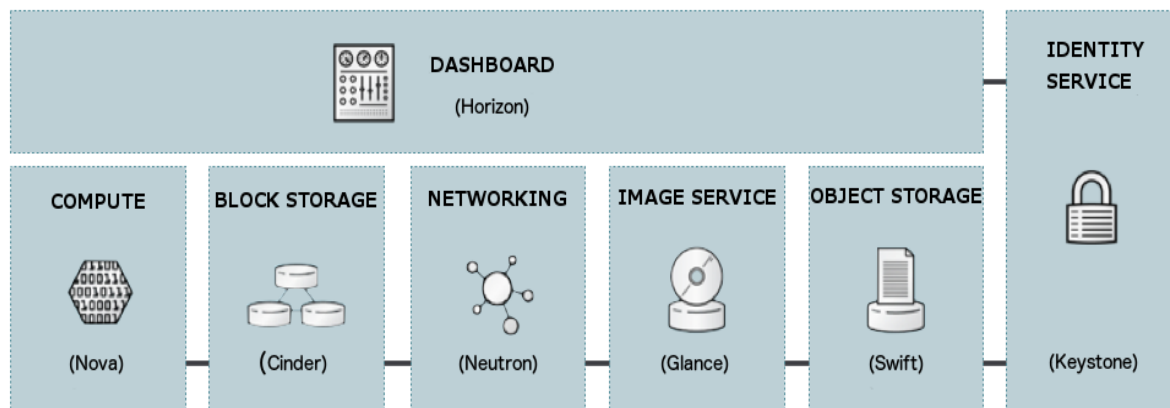


Figure 24: A high-level view of OpenStack core services.

OpenStack is a rich and complex software environment with numerous capabilities for cloud management. Within ACTiCLOUD, and for the purposes of the ACTiManager, the following OpenStack core components / services are relevant to our approach and we have integrated support for them within the MicroVisor platform:

1. **Nova**<sup>8</sup> is the primary computing engine behind OpenStack. It is used for deploying and managing large numbers of virtual machines and other instances to handle computing tasks. This module is required to spawn new VMs, migrate VMs, terminate VMs, and other important operations needed by the ACTiManager.

<sup>8</sup> <https://docs.OpenStack.org/developer/nova/>

2. **Cinder** is a block storage service that provides persistent block storage for compute instances. This service is responsible for managing the life-cycle of block devices, from the creation and attachment of volumes to instances, to their release.

3. **Neutron** is OpenStack's networking service that provides various networking services to cloud users (tenants) such as IP address management, DNS, DHCP, load balancing, and security groups (network access rules, like firewall policies). The Neutron service provides a framework for software defined networking (SDN) that allows for pluggable integration with various networking solutions.

4. **Keystone** is OpenStack's shared identity service that provides authentication and authorization services throughout the entire cloud infrastructure. The Keystone service has pluggable support for multiple forms of authentication.

5. **Glance** is OpenStack's Image service that provides disk-image management services, including image discovery, registration, and delivery services to the Compute service, as needed.

Besides the above core services, OpenStack requires the use of messaging for internal communication between several services. By default, OpenStack uses message queues based on the AMQP, which, like most OpenStack services, supports pluggable components. In our current implementation on the MicroVisor we are using the default RabbitMQ for the messaging backend.

As we provide support for the core services mentioned above, there are several additional services that operate on top of the core services, required by the ACTiManager layer, that can be supported without extra drivers for the MicroVisor virtualization platform. These are:

**Horizon** is OpenStack's dashboard, providing a web-based interface (UI) for both cloud administrators and cloud tenants, in order to provision, manage, and monitor cloud resources.

**Heat**<sup>9</sup> is the orchestration component of OpenStack, which allows developers to store the requirements of a cloud application in a file that defines what resources are necessary for that application. In this way, it helps to manage the infrastructure needed for a cloud service to run. This module can be extended to accommodate policies related to applications' scaling (scale up/down, scale out/in) requirements.

## 4.2 Revised OpenStack architecture on the MicroVisor platform

### 4.2.1 Implementation of OpenStack Pike in the previous prototypes

In the ACTiCLOUDStrawman (first) prototype we used the Kilo version of OpenStack, which was integrated into the MicroVisor Controller node as a management and orchestration layer. Continuing our work for the ACTiCLOUD Intermediate (second) prototype, we have ported the drivers of the Kilo version of OpenStack to the recent version of OpenStack named "Pike"<sup>10</sup>. The Pike version includes updates focusing on manageability, flexibility and scale<sup>11</sup>. In the Intermediate prototype OnApp has provided support and ported drivers for integration with the MicroVisor for this new OpenStack version.

However, although the platform components (hypervisor, virtxd, distributed storage stack, web-based UI, OpenStack Pike drivers) have been implemented in the Intermediate prototype, there

---

<sup>9</sup> <https://wiki.OpenStack.org/wiki/Heat>

<sup>10</sup> <https://releases.OpenStack.org/pike/index.html>

<sup>11</sup> <https://www.OpenStack.org/software/pike/>

were instabilities requiring more testing and bug-fixes in the management layer. More importantly, it should be noted that the implemented version of the OpenStack drivers for the MicroVisor has been based on the logic of the Kilo version, which did not include the new concepts and abstractions introduced in later versions until Pike (e.g. cells, resource groups, etc.). In our efforts for the Final prototype we have strived for a redesign of the OpenStack drivers, in order to allow better and tighter integration with the concepts and APIs that are supported in the Pike version.

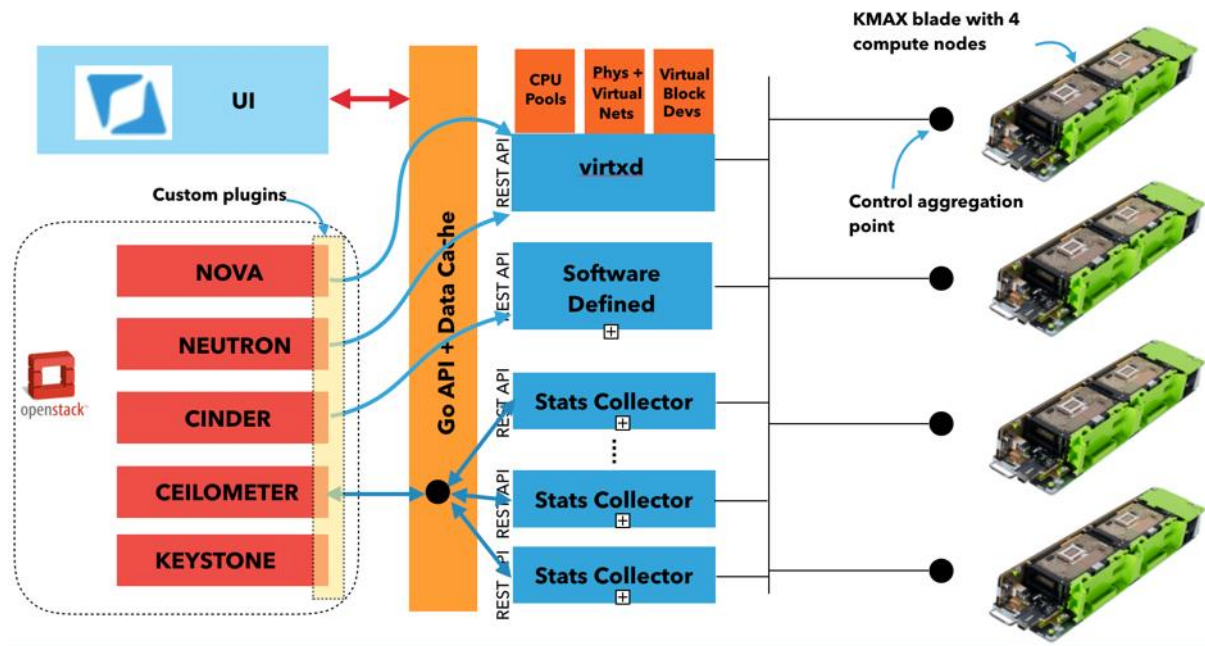


Figure 25: OpenStack - MicroVisor Architecture

A high-level view of the management layer architecture of the Intermediate prototype on the KMAX platform is shown in Figure 21. As depicted, the OpenStack services are tightly coupled with the MicroVisor services for the platform management, hypervisor control, monitoring and the UI and all these services were running on one controller node VM.

During our efforts to improve stability and implement redesigned OpenStack drivers, allowing better and tighter integration with the concepts and APIs of the Pike version, we have realized that the tight integration of OpenStack services with the MicroVisor management and monitoring services, not only increased significantly the complexity of the implementation and the instability, but also required that one could only control the infrastructure through OpenStack (e.g. via Horizon UI) and not through our MicroVisor API, CLI and web-based UI. Furthermore, in our effort to support both control layers at the same time, we had to maintain system and resource status simultaneously in two components (i.e. in two databases) and also provide limited support for OpenStack Pike services that are required for the ACTiManager integration.

Upon realizing these important limitations we decided to reconsider our design and decouple the OpenStack services from the MicroVisor management layer, resulting to a new management architecture for the Final ACTiCLOUD prototype, as presented next.

#### 4.2.2 A new OpenStack management architecture for the Final prototype

Rethinking the management layer architecture of the MicroVisor with unrestricted OpenStack Pike support for the OpenStack API and the services required for the ACTiManager we have decided to redesign and implement a new cleaner architecture. This architecture splits the “internal” MicroVisor management layer in the controller node from the OpenStack control stack and services, which would be externalized and run on any node or VM, local to the cluster or even remote over a TCP/IP connection, through a separate “external API” of the MicroVisor. In this redesigned architecture, the user would be able to control the infrastructure either through the MicroVisor CLI, API or UI, or through OpenStack, but not through both control layers simultaneously.

With this separation of components through the “external API”, the new management architecture of the MicroVisor would look like the following Figure 22, in contrast to Figure 21. In this figure we observe that the internal MicroVisor controller is contained within a controller node with all the required services and exposing two different APIs, one internal API that is used by the native MicroVisor UI (denoted “internal REST API”), and one external API that can be used by external orchestration platforms to control the MicroVisor cluster.

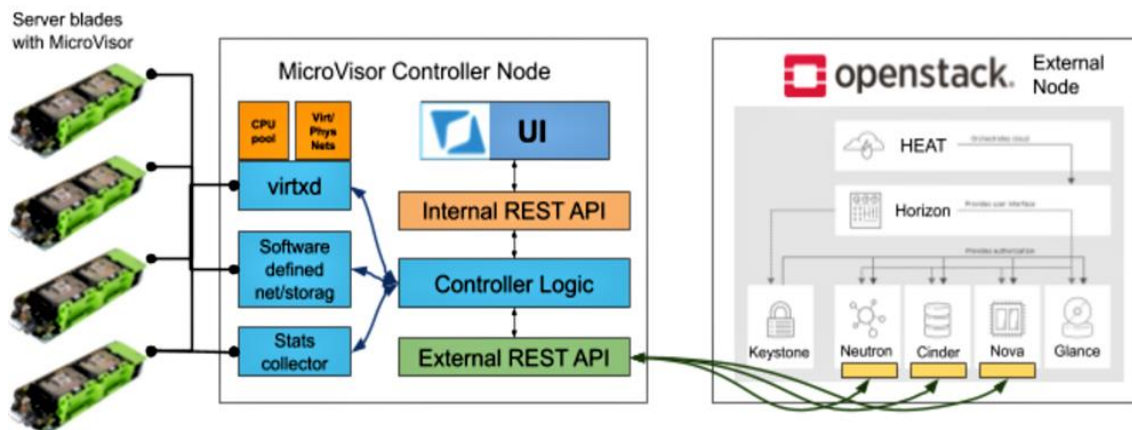


Figure 26: New architecture for OpenStack on the MicroVisor platform

One additional advantage of separating the OpenStack controller from the internal MicroVisor controller and by providing control through the “external API” is that these drivers we provide can be used by a standard, unmodified OpenStack distribution, such as a Devstack distribution of Pike<sup>12</sup>. That would allow drivers to be easily installed in a standard OpenStack installation and also to port and maintain such drivers for later OpenStack releases.

### 4.3 OpenStack Implementation for the Final Prototype

The implementation of the rack-scale MicroVisor for the Final prototype provides support for the stable version of OpenStack named “Pike”<sup>13</sup>. Pike version includes updates focusing on

<sup>12</sup> <https://docs.OpenStack.org/devstack/pike/>

<sup>13</sup> <https://releases.OpenStack.org/pike/index.html>

manageability, flexibility and scale. OnApp has provided support and ported the required drivers for the integration with the MicroVisor for this new OpenStack version through the external API. Thus, implementing Pike for the final prototype improves our initial integration effort completed for the Strawman prototype and also maximizes the impact of ACTiCLOUD to the relevant communities and industries.

The external OpenStack implementation discussed in Section 5.2 and depicted in Figure 22, has significant differences from the previous implementation and required several updates in existing components and the creation of new components. In brief, the components that are new or have updates are listed and discussed below:

1. A new “**external API**” for the MicroVisor controller node.
2. OpenStack drivers for the three core services: Nova, Cinder and Neutron. Other OpenStack services build on top of these to provide higher-level services, such as Heat.
3. Installation scripts to install these drivers on an unmodified standard OpenStack.

The updates on these components are discussed in detail in the following sections.

#### 4.3.1 External API

We have implemented a new “external API” for the MicroVisor controller, which operates on different endpoints than the internal one to facilitate external operations and can be used to control the MicroVisor cluster remotely. The external API for the OpenStack Pike is the first such implementation and it is documented extensively in Appendix I, Section 7.1.

Note that the External API does not support all features of the MicroVisor, as the internal API. Instead, the external API is adapted to the features and requirements of the external management service that will be controlling the MicroVisor cluster, which is OpenStack Pike in this case.

The REST API endpoints are presented extensively in Appendix I, Section 7.1, including the request type, parameters, responses and error responses. The current API controls four different types of objects: (a) Nodes, (b) Instances, (c) Storage Volumes and (d) Networks.

##### 4.3.1.1 Configuration for the External API

To use the external API, one must first initialize the MicroVisor cluster using the native MicroVisor UI or CLI. The necessary configuration steps are:

1. Set the authentication (password) for external API access. This is required since all calls to the API endpoints are authenticated.
2. Create a datastore (with a specific name) with all the storage devices that will be managed through the external API and OpenStack. It is possible to exclude devices from this datastore, which can then be managed only via the native MicroVisor UI or CLI.
3. Create a resource group (with a specific name) for any resources managed through the external API and OpenStack. Currently our implementation supports a single resource group, however we plan to extend support for multiple resource groups created and managed through OpenStack.

After these important steps the external API is available for use.



### 4.3.2 OpenStack drivers

The second part of the implementation consists of the drivers for the core OpenStack services, Nova, Cinder and Neutron, discussed below. The rest of the OpenStack services do not require drivers for the hypervisor platform, because they are either self-contained (e.g. Keystone and Glance), or they are based on top of the core services (e.g. Heat, Horizon, Watcher, etc.).

#### 4.3.2.1 Compute: Nova

The Nova module is used for deploying and managing VMs and other instances and runs as a set of daemons on top of existing Linux servers. Nova requires the following additional OpenStack services for basic function: (i) Keystone, that provides identity and authentication for all OpenStack services, (ii) Glance that provides the compute image repository, (iii) Neutron, that is responsible for provisioning the virtual or physical networks that compute instances connect to on boot. Finally, Nova can also integrate with other services to include: persistent block storage, encrypted disks, and bare metal compute instances.

A custom Nova compute plugin has been developed for the MicroVisor platform, allowing the Nova service to communicate with and control the MicroVisor infrastructure via the external API, in order to manage hypervisor nodes and VMs. Functions such as provision, start, stop, pause, resume, and delete a VM are all mapped through the MicroVisor external API to the controller logic which manages the MicroVisor instances.

#### 4.3.3 Network: Neutron

Neutron is the networking service for OpenStack, which provides various networking services to instances, such as IP address management, DNS, DHCP, and security groups (network access rules, like firewall policies). The Neutron service provides a framework for software defined networking (SDN) that allows for pluggable integration with various networking solutions.

Support for Neutron networking service has been included in the Final prototype, currently supporting a simple physical network virtual interface for instances with IP address configuration managed through OpenStack. The Neutron support will be further extended in the plugin driver to support more advanced network configurations, as well as routers for virtual networks that are provided by the MicroVisor platform.

#### 4.3.3.1 Storage: Cinder

Cinder is a block storage service for OpenStack, designed to present storage resources to end users that can be consumed by Nova. This is accomplished through the use of either a reference implementation or plugin drivers for other storage. Cinder virtualizes the management of block storage devices and provides end users with a self-service API to request and consume those resources without requiring any knowledge of where their storage is actually deployed or on what type of device.

The MicroVisor platform includes a complete Software Defined Storage (SDS) layer that enables mapping of locally attached storage device capacity (NVMe, SSD, SATA etc.) as virtualized drives to a virtual machine. The SDS layer is a block volume manager that creates distributed block storage volumes, complete with redundancy for fault tolerance and locality awareness to minimize access latency for block IO requests.

In order to support the MicroVisor block volume manager, a custom OpenStack plugin driver was also implemented for integration with Cinder. The volume manager is controlled via an API agent that runs locally on the OpenStack controller node as well as by communicating to the MicroVisors directly to map virtual block devices via the embedded ATA-over-Ethernet (AoE) client directly to a virtual machine guest. The active block path involves a cooperative frontend (MicroVisor AoE client) and backend (Storage service that directly controls the local physical storage devices). The frontend is responsible for mirroring and failover of paths in the event of failure, whilst the backend is responsible for maintaining consistency across replicas of data as well as serving the active block IO requests.

#### 4.3.4 Installation scripts

For the purposes of the OpenStack integration we have also developed a few installation scripts, facilitating an easy installation of the plugin drivers to a standard unmodified OpenStack distribution, such as the Devstack distribution of Pike<sup>14</sup>. The installation scripts are written in Python and also facilitate the installation of OpenStack Pike on a new standard Ubuntu 18 host or VM, with a minimum requirement of 6GB of memory and 40GB storage.

#### 4.3.5 Open-source release of OpenStack drivers

To increase the impact and visibility of our work on OpenStack integration, OnApp will be releasing these drivers as open-source code with the Apache version 2 license<sup>15</sup> in the following public bitbucket repository: [https://bitbucket.org/sunlightio/OpenStack\\_integration/](https://bitbucket.org/sunlightio/OpenStack_integration/). In this git-based repository we have added the current version of the OpenStack drivers, installation scripts, documentation of the external API, as well as the installation process and scripts. In the same public repository we will push any new updates, bugfixes and new releases providing support for more recent OpenStack versions.

### 4.4 Control via OpenStack's Horizon UI

OpenStack Horizon is a web-based graphical interface that is used for managing OpenStack compute, storage and networking services. Through the Horizon UI, we are able to launch virtual machine instances, view the size and current state of their OpenStack cloud deployment, manage networks, and set limits on the cloud resources available. In general, Horizon acts as a self-service portal to provision cloud resources. As mentioned in previous sections, the Horizon interface works with OpenStack Nova, a compute service; OpenStack Cinder, a block storage system; OpenStack Keystone, an identity management service; and many others. It accesses these services via OpenStack application programming interfaces (APIs).

The Horizon UI offers three versions of management dashboards: a User Dashboard, a System Dashboard and a Settings Dashboard. The ability to customize visual elements of the Horizon interface is also provided, including the navigation bar, tables, alerts and other elements, or even build applications that integrate with the dashboard. In addition, it is possible to integrate third-party management and monitoring tools with OpenStack Horizon<sup>16</sup>.

---

<sup>14</sup> <https://docs.OpenStack.org/devstack/pike/>

<sup>15</sup> <https://www.apache.org/licenses/LICENSE-2.0>

<sup>16</sup> <https://docs.OpenStack.org/horizon/latest/index.html>



In the Final prototype we have tested the OpenStack Horizon UI dashboard running on the OpenStack controller node, working to control the MicroVisor cluster via the external API. Note, however, that although we have set up and used the Horizon UI in our OpenStack controller, one can also use the MicroVisor native UI, which currently provides a superset of features compared to the Horizon UI, such as configuring better resource-control features.

The screenshots displayed below depict this Horizon UI integration and provide a visual representation of the implementation of the Final prototype.

#### 4.4.1 Listing Hypervisors in the MicroVisor cluster

The screen below shows two hypervisors available for creating instances.

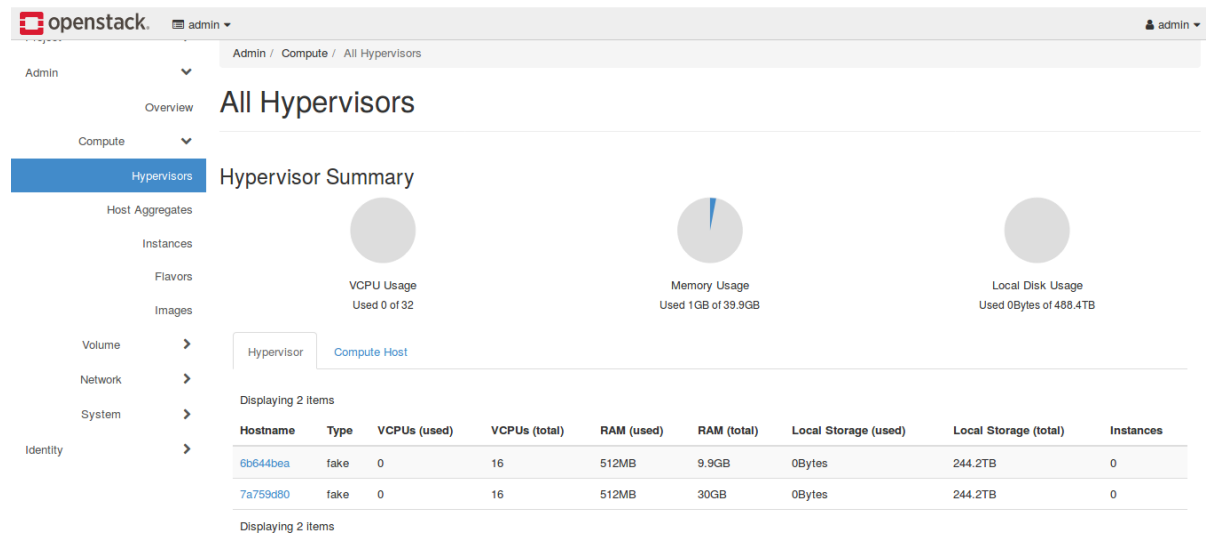


Figure 27: List of Hypervisors

#### 4.4.2 Creating Instances

Creating an instance requires several details to be provided, as demonstrated below.

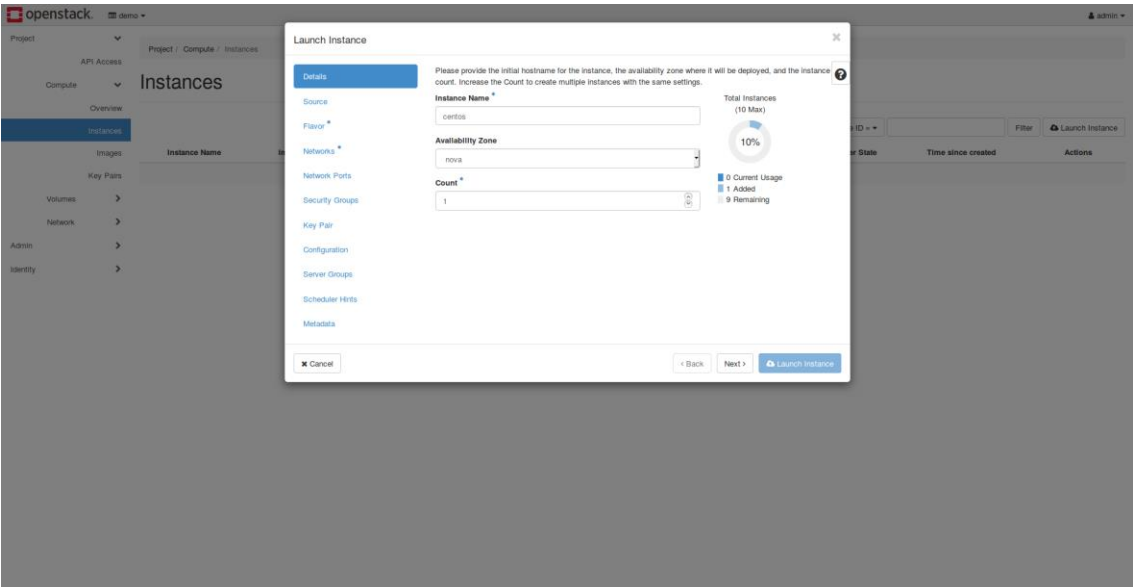


Figure 28: Create instance details

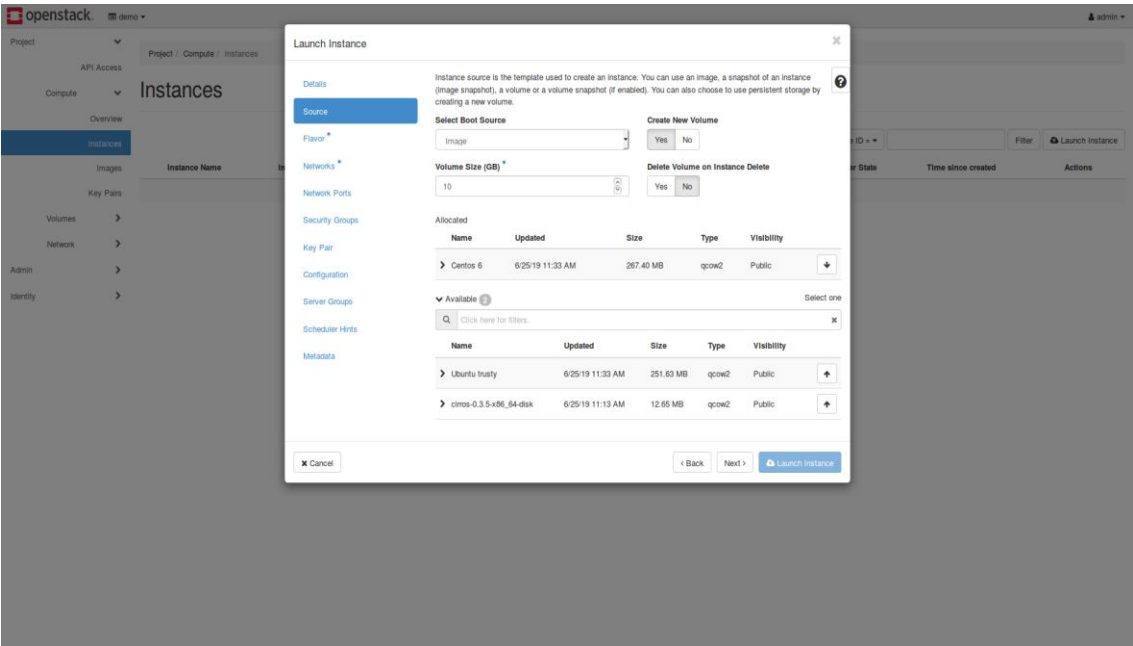


Figure 29: Create instance source

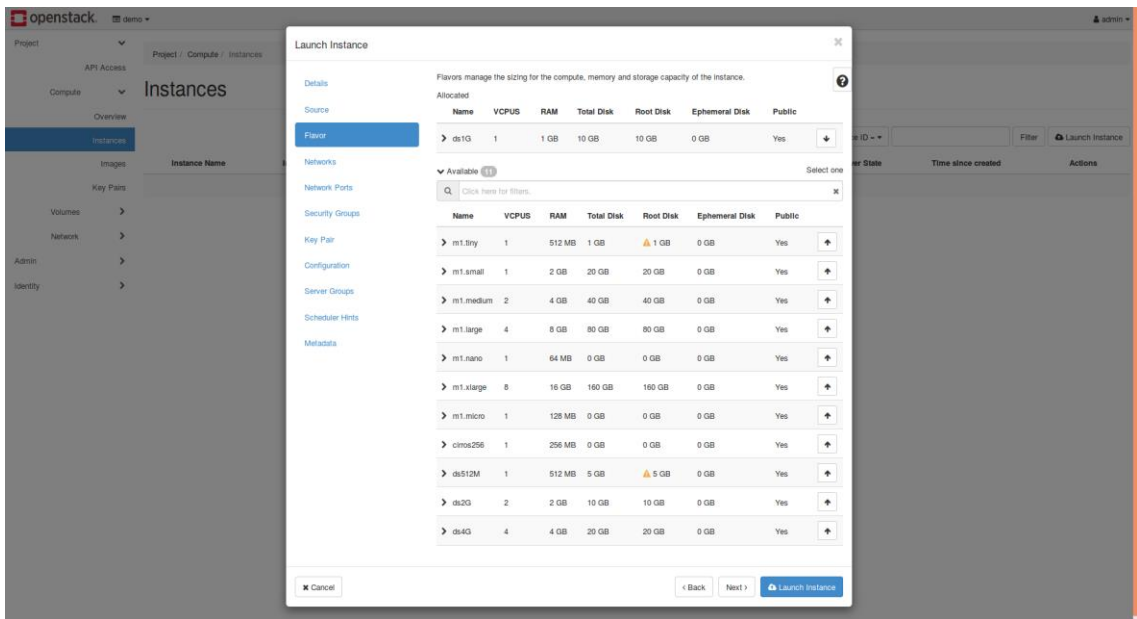


Figure 30: Create instance flavor

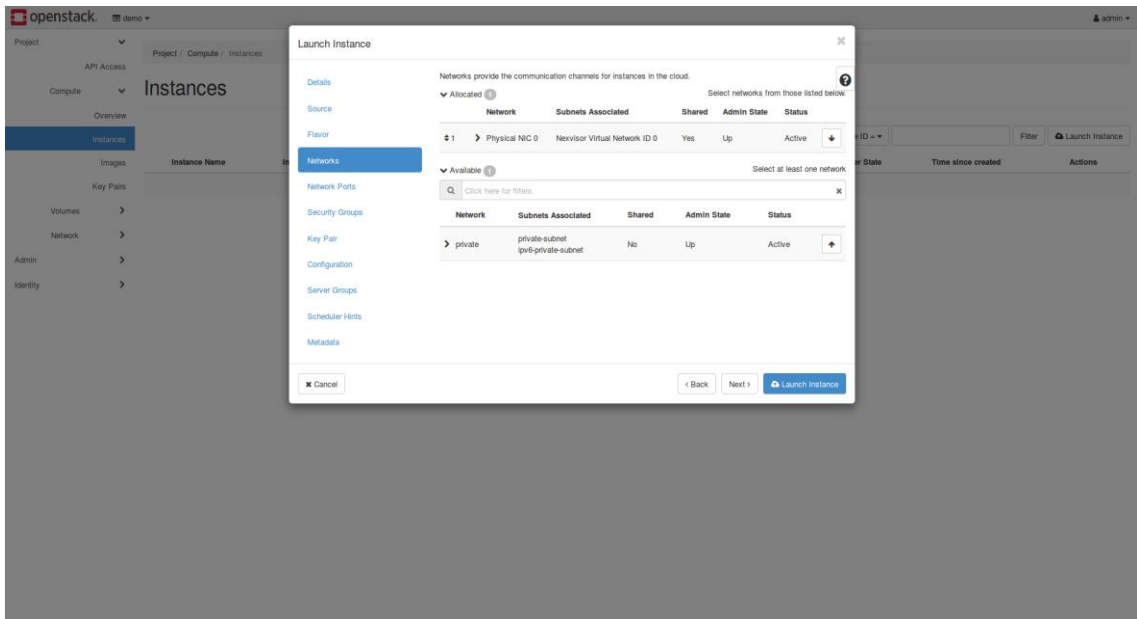


Figure 31: Create instance networks

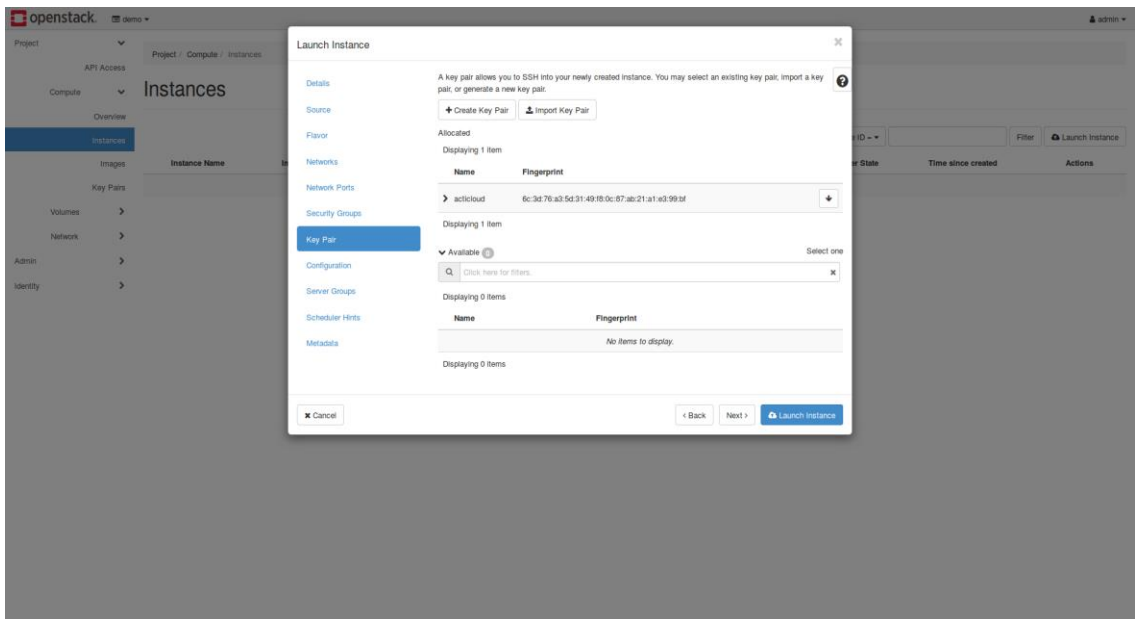


Figure 32: Create instance key pair

Following the creation, an instance is displayed below.

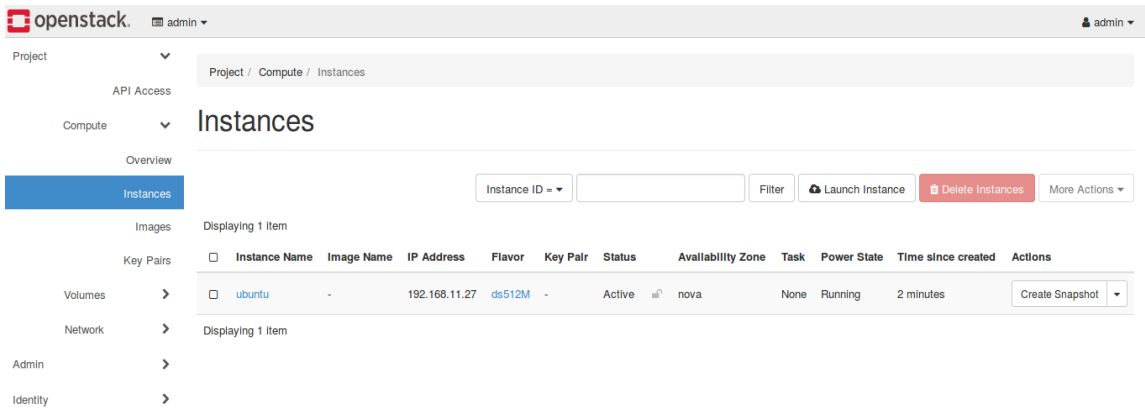


Figure 33: New instance created

4.4.3 Creating Flavors

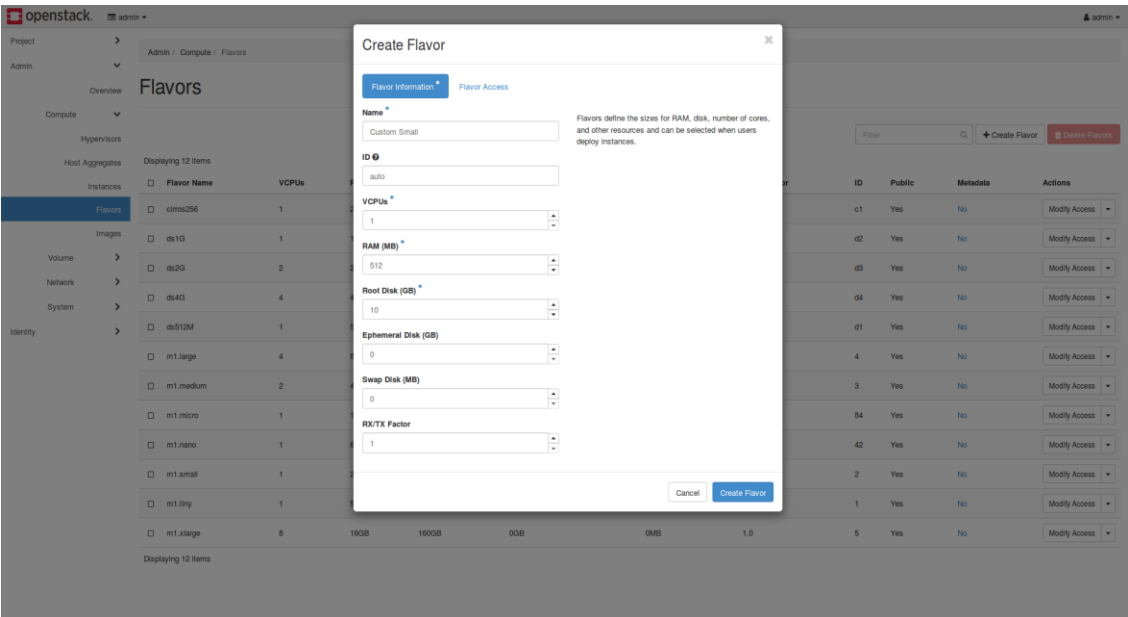


Figure 34: Create flavor form

## 5 Conclusions

This deliverable D2.3 describes the design and implementation of the final prototype of the rack-scale MicroVisor, as well as the integration with OpenStack platform in ACTiCLOUD

It presents the updates that have been performed since the ACTiCLOUD Intermediate (second) prototype, regarding the implementation and integration of the MicroVisor. Following the analysis of the rack-scale MicroVisor architecture, including monitoring and performance features, this deliverable also analysed the important aspect of resource control through the MicroVisor's concept of "resource groups".

Through the presentation of several features that have been implemented, we believe that the final prototype has achieved the goals of the ACTiCLOUD project, providing great value to the CSPs and the end users.

## 6 Appendix I

### 6.1 MicroVisor External API for OpenStack Pike

The external API for OpenStack drivers.

Version: 0.1

BasePath:/slapi

*All rights reserved*

<http://apache.org/licenses/LICENSE-2.0.html>

#### 6.1.1 Instances

**PUT /instances/{instance\_name}/action**

*Execute an action on the instance. (instancesInstanceNameActionPut)*

*Path parameters*

instance\_name (required)

Path Parameter — *The instance name.*

#### Consumes

This API call consumes the following media types via the Content-Type request header:

- application/json

#### Request body

**instance NewInstanceAction (optional)**

*Body Parameter — The action on instance.*

#### Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- application/json

#### Responses

##### 202

The action executed successfully.

##### 404

The resource does not exist. Error

#### Example data

Content-Type: application/json

```
{code=404, message=Resource with id 1123 does not exist.}
```

##### 422

One or more validations have failed during a request processing. Error

#### Example data

Content-Type: application/json

```
{code=422, message=Validation failed}
```

**500**

An unexpected error occurred. Error

**Example data**

Content-Type: text/plain

```
{code=500, message=Operation X did something wrong during processing}
```

**GET /instances/{instance\_name}/info**

Get instance's information (**instancesInstanceNameInfoGet**)

This request will send information about the state of the instance and some hardware information.

**Path parameters****instance\_name (required)**

*Path Parameter* — The instance name.

**Consumes**

This API call consumes the following media types via the Content-Type request header:

- application/json

**Return type**

InstanceInformation

**Example data**

Content-Type: application/json

```
{
  "mem_kb" : 1,
  "cpu_time_ns" : 5,
  "state" : 0,
  "max_mem_kb" : 6,
  "num_cpu" : 5
}
```

**Produces**

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- application/json

**Responses****200**

Instance's information. InstanceInformation

**500**

An unexpected error occurred. Error

**Example data**

Content-Type: text/plain



```
{code=500, message=Operation X did something wrong during processing}
```

## POST /instances

Create a new instance (**instancesPost**)

### Consumes

This API call consumes the following media types via the Content-Type request header:

- application/json

### Request body

#### instance NewInstance (optional)

*Body Parameter* — The instance to create.

### Return type

Instance

### Example data

Content-Type: application/json

```
{
  "admin_ui_link" : "admin_ui_link",
  "swarm_node_role" : "worker",
  "resource_group_id" : 1,
  "is_storage_latency_optimized" : "fully",
  "networks" : [ {
    "network_id" : 4,
    "interface_id" : 2,
    "ip" : "ip",
    "name" : "name",
    "mac" : "mac"
  }, {
    "network_id" : 4,
    "interface_id" : 2,
    "ip" : "ip",
    "name" : "name",
    "mac" : "mac"
  } ],
  "is_leader" : true,
  "domain_id" : 6,
  "domain_name" : "domain_name",
  "hostname" : "hostname",
  "cluster_id" : "046b6c7f-0b8a-43b9-b35d-6489e6daee91",
  "vm_status" : "vm_status",
  "id" : 0,
  "state" : "ACTIVE",
  "vm_cpus" : 3,
```

```

"ram" : 1,
"cluster_type_id" : "swarm",
"owner" : "owner",
"swap" : 7,
"vm_memory_mb" : 9,
"OpenStack_uuid" : "046b6c7f-0b8a-43b9-b35d-6489e6daee91",
"volumes" : [ {
  "is_degraded" : true,
  "instance_id" : 6,
  "size" : 1,
  "disks" : [ "disks", "disks" ],
  "is_repairing" : true,
  "name" : "name",
  "datastore_id" : "datastore_id",
  "vdisk_id" : "vdisk_id",
  "id" : 0,
  "percent_repair" : 5,
  "status" : "in-use"
}, {
  "is_degraded" : true,
  "instance_id" : 6,
  "size" : 1,
  "disks" : [ "disks", "disks" ],
  "is_repairing" : true,
  "name" : "name",
  "datastore_id" : "datastore_id",
  "vdisk_id" : "vdisk_id",
  "id" : 0,
  "percent_repair" : 5,
  "status" : "in-use"
} ],
"storage_optimized" : true,
"vcpus" : 1,
"node" : "node",
"disk" : 1,
"name" : "name"
}

```

### Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- application/json

### Responses

#### 201

Instance successfully created. Instance

**400**

Status bad request. Error

**Example data**

Content-Type: application/json

```
{code=400}
```

**404**

The resource does not exist. Error

**Example data**

Content-Type: application/json

```
{code=404, message=Resource with id 1123 does not exist.}
```

**422**

One or more validations have failed during a request processing. Error

**Example data**

Content-Type: application/json

```
{code=422, message=Validation failed}
```

**500**

An unexpected error occurred. Error

**Example data**

Content-Type: text/plain

```
{code=500, message=Operation X did something wrong during processing}
```

---

**6.1.2 Networks****GET /networks**Get all the networks. (**networksGet**)**Consumes**

This API call consumes the following media types via the Content-Type request header:

- application/json

**Return type**array[[Network](#)]**Example data**

Content-Type: application/json

```
[ {
  "ip_end" : "ip_end",
  "active_nodes" : [ "active_nodes", "active_nodes" ],
  "is_private" : true,
  "virtual" : true,
  "is_vlan" : false,
  "dhcp_type" : "internal",
```

```

"vnet_id" : "vnet_id",
"is_routable" : false,
"resource_group_ids" : [ 6, 6 ],
"mac" : "mac",
"dhcp_server_ip" : "dhcp_server_ip",
"physical_paths" : [ 5, 5 ],
"vlan_tag" : 5,
"is_dedicated" : true,
"ip_version" : 1,
"ip_start" : "ip_start",
"name" : "name",
"cidr" : "cidr",
"id" : 0,
"gateway" : "gateway",
"nat_forwarding" : false,
"is_wire_vlan" : false
}, {
"ip_end" : "ip_end",
"active_nodes" : [ "active_nodes", "active_nodes" ],
"is_private" : true,
"virtual" : true,
"is_vlan" : false,
"dhcp_type" : "internal",
"vnet_id" : "vnet_id",
"is_routable" : false,
"resource_group_ids" : [ 6, 6 ],
"mac" : "mac",
"dhcp_server_ip" : "dhcp_server_ip",
"physical_paths" : [ 5, 5 ],
"vlan_tag" : 5,
"is_dedicated" : true,
"ip_version" : 1,
"ip_start" : "ip_start",
"name" : "name",
"cidr" : "cidr",
"id" : 0,
"gateway" : "gateway",
"nat_forwarding" : false,
"is_wire_vlan" : false
} ]

```

### Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- application/json

**Responses****200**

A list of networks.

**500**

An unexpected error occurred. Error

**Example data**

Content-Type: text/plain

```
{code=500, message=Operation X did something wrong during processing}
```

**6.1.3 Nodes****GET /nodes**Get the IDs from all the nodes. (**nodesGet**)**Consumes**

This API call consumes the following media types via the Content-Type request header:

- application/json

**Return type**

ListNodes

**Example data**

Content-Type: application/json

```
{
  "nodes": [ "nodes", "nodes" ]
}
```

**Produces**

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- application/json

**Responses****200**

Successfully got list of nodes. ListNodes

**500**

An unexpected error occurred. Error

**Example data**

Content-Type: text/plain

```
{code=500, message=Operation X did something wrong during processing}
```

**GET /nodes/{node\_id}/resources**Get the resources information for a node. (**nodesNodeIdResourcesGet**)**Path parameters****node\_id (required)**

*Path Parameter* — The node id.

### Consumes

This API call consumes the following media types via the Content-Type request header:

- application/json

### Return type

#### Resources

### Example data

Content-Type: application/json

```
{
  "local_gb_used" : 2.3021358869347654518833223846741020679473876953125,
  "local_gb" : 1.46581298050294517310021547018550336360931396484375,
  "memory_mb_used" : 5.63737665663332876420099637471139430999755859375,
  "disk_available_least" : 7.061401241503109105224211816675961017608642578125,
  "vcpus_used" : 5.962133916683182377482808078639209270477294921875,
  "memory_mb" : 6.02745618307040320615897144307382404804229736328125,
  "vcpus" : 0.80082819046101150206595775671303272247314453125
}
```

### Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- application/json

### Responses

#### 200

Successfully got resources information for a node. Resources

#### 404

The resource does not exist. Error

### Example data

Content-Type: application/json

```
{code=404, message=Resource with id 1123 does not exist.}
```

#### 500

An unexpected error occurred. Error

### Example data

Content-Type: text/plain

```
{code=500, message=Operation X did something wrong during processing}
```

## 6.1.4 Volumes

### POST /volumes

Create a new volume. (**volumesPost**)

Create a new empty volume. The following scenarios will fail:

- If name is empty (422)
- If size is 0 or less (422)

### Consumes

This API call consumes the following media types via the Content-Type request header:

- application/json

### Request body

#### volume **NewVolume** (optional)

*Body Parameter* — The volume to create.

### Return type

#### Volume

#### Example data

Content-Type: application/json

```
{
  "is_degraded" : true,
  "instance_id" : 6,
  "size" : 1,
  "disks" : [ "disks", "disks" ],
  "is_repairing" : true,
  "name" : "name",
  "datastore_id" : "datastore_id",
  "vdisk_id" : "vdisk_id",
  "id" : 0,
  "percent_repair" : 5,
  "status" : "in-use"
}
```

### Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- application/json

### Responses

#### 201

Volume successfully created. Volume

#### 422

One or more validations have failed during a request processing. Error

#### Example data

Content-Type: application/json

```
{code=422, message=Validation failed}
```

#### 500

An unexpected error occurred. Error

**Example data**

Content-Type: text/plain

```
{code=500, message=Operation X did something wrong during processing}
```

**PUT /volumes/{volume\_name}/clone**Clone a volume (**volumesVolumeNameClonePut**)

It will create a new volume based on a specified volume.

**Path parameters****volume\_name (required)***Path Parameter* — The volume name.**Consumes**

This API call consumes the following media types via the Content-Type request header:

- application/json

**Request body****volume NewVolume (optional)***Body Parameter* — The volume to create.**Return type**

Volume

**Example data**

Content-Type: application/json

```
{
  "is_degraded" : true,
  "instance_id" : 6,
  "size" : 1,
  "disks" : [ "disks", "disks" ],
  "is_repairing" : true,
  "name" : "name",
  "datastore_id" : "datastore_id",
  "vdisk_id" : "vdisk_id",
  "id" : 0,
  "percent_repair" : 5,
  "status" : "in-use"
}
```

**Produces**

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- application/json

**Responses****201**



Volume successfully created. Volume

**404**

The resource does not exist. Error

**Example data**

Content-Type: application/json

```
{code=404, message=Resource with id 1123 does not exist.}
```

**500**

An unexpected error occurred. Error

**Example data**

Content-Type: text/plain

```
{code=500, message=Operation X did something wrong during processing}
```

**DELETE /volumes/{volume\_name}**

Delete volume. (**volumesVolumeNameDelete**)

**Path parameters****volume\_name (required)**

*Path Parameter* — The volume name.

**Consumes**

This API call consumes the following media types via the Content-Type request header:

- application/json

**Produces**

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- application/json

**Responses****204**

Volume deleted.

**404**

The resource does not exist. Error

**Example data**

Content-Type: application/json

```
{code=404, message=Resource with id 1123 does not exist.}
```

**422**

One or more validations have failed during a request processing. Error

**Example data**

Content-Type: application/json

```
{code=422, message=Validation failed}
```

**500**

An unexpected error occurred. Error

**Example data**

Content-Type: text/plain

```
{code=500, message=Operation X did something wrong during processing}
```

**DELETE /volumes/{volume\_name}/nbd**

Disable NBD port for the specific volume (**volumesVolumeNameNbdDelete**)

The controller will close the port which the specific volume is using.

**Path parameters****volume\_name (required)**

*Path Parameter* — The volume name.

**Consumes**

This API call consumes the following media types via the Content-Type request header:

- application/json

**Produces**

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- application/json

**Responses****200**

NBD port successfully disabled.

**404**

The resource does not exist. Error

**Example data**

Content-Type: application/json

```
{code=404, message=Resource with id 1123 does not exist.}
```

**500**

An unexpected error occurred. Error

**Example data**

Content-Type: text/plain

```
{code=500, message=Operation X did something wrong during processing}
```

**POST /volumes/{volume\_name}/nbd**

Enable NBD port for the specific volume (**volumesVolumeNameNbdPost**)

This request will open a port on the controller for the OpenStack to use it through the nbd-client.

**Path parameters****volume\_name (required)**

*Path Parameter* — The volume name.

**Consumes**

This API call consumes the following media types via the Content-Type request header:

- application/json

**Return type**

NbdPort

**Example data**

Content-Type: application/json

```
{
  "port" : 0.80082819046101150206595775671303272247314453125
}
```

**Produces**

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- application/json

**Responses****200**

NBD port successfully created. NbdPort

**404**

The resource does not exist. Error

**Example data**

Content-Type: application/json

```
{code=404, message=Resource with id 1123 does not exist.}
```

**500**

An unexpected error occurred. Error

**Example data**

Content-Type: text/plain

```
{code=500, message=Operation X did something wrong during processing}
```

---

## 6.2 Models

### 6.2.1 Table of Contents

1. [Error](#) -
2. [Instance](#) -
3. [InstanceInformation](#) -
4. [InstanceNetworkInterface](#) -
5. [ListNodes](#) -
6. [NbdPort](#) -
7. [Network](#) -
8. [NewInstance](#) -
9. [NewInstanceAction](#) -

- 10. NewInstanceIP -
- 11. NewInstanceVolume -
- 12. NewVolume -
- 13. Resources -
- 14. Volume -

## 1. Error

### errors (optional)

array[String] A list of human-readable messages, describing the errors.

## 2. Instance

### id (optional)

Long format: int64

### name (optional)

String The name of the instance.

### domain\_id (optional)

Long The ID of the VM in the NexVisor. It is useful for the administrator. format: int64

### domain\_name (optional)

String The name of the VM in the NexVisor. It is useful for the administrator.

### node (optional)

String The id of the node where the instance is located.

### resource\_group\_id (optional)

Long The id of the resource group that the instance makes use of. format: int64

### state (optional)

String

#### Enum:

ACTIVE

BUILD

PAUSED

STOPPED

ERROR

PAUSING

UNPAUSING

POWERING-OFF

POWERING-ON

SPAWNING

DELETING

UPDATING

### owner (optional)

String The ID of the user.

### hostname (optional)

String The hostname of the instance.

### disk (optional)

Integer The volume's size of the instance in GB. format: int32

### ram (optional)

Integer The memory's size of the instance in MB. format: int32

### vcpus (optional)

Integer The number of virtual cores the instance has to use. format: int32

**swap (optional)**

Integer The disk space of the instance in GB. format: int32

**cluster\_id (optional)**

UUID The id of the cluster that this instance belongs to. format: uuid

**admin\_ui\_link (optional)**

String This property is present only if the instance belongs to a docker or docker swarm cluster. In addition to the latter case, only if it is marked as a manager instance.

**OpenStack\_uuid (optional)**

UUID The id of the instance based on the OpenStack. format: uuid

**vm\_status (optional)**

String The state of the VM based on the Virtxd. If the instance is closed it will be empty.

**vm\_memory\_mb (optional)**

Long The memory of the VM based on the Virtxd. If the instance is closed it will be 0. format: int64

**vm\_cpus (optional)**

Long The number of CPUs of the VM based on the Virtxd. If the instance is closed it will be 0. format: int64

**networks (optional)**

array[InstanceNetworkInterface] The list of network interfaces connected to the instance. If a network interface does not have an IP then it is based on a physical network.

**volumes (optional)**

array[Volume] The list of virtual disks attached to the instance.

**cluster\_type\_id (optional)**

String If the instance is in a cluster, then it shows its cluster type.

**Enum:**

swarm

vm

docker

**storage\_optimized (optional)**

Boolean When the instance has volumes with storages from the same node.

**is\_storage\_latency\_optimized (optional)**

String

The state of the storage optimization that the instance has currently. Here are the meaning of each value:

- 'fully' means that all the virtual disks of the instance are storage optimized
- 'partially' means that some of the virtual disks of the instance are storage optimized
- 'none' means that none of the virtual disks of the instance are storage optimized

**Enum:**

fully

partially

none

**is\_leader (optional)**

Boolean If the instance is the leader of the cluster

**swarm\_node\_role (optional)**

String The role of the instance if it is in a cluster of type 'swarm'.

**Enum:**

worker

manager

**3. InstanceInformation**

**state (optional)***Integer* The state of the instance.**max\_mem\_kb (optional)***Integer* The limit of instance's memory.**mem\_kb (optional)***Integer* The current use of the instance's memory.**num\_cpu (optional)***Integer* The current number of CPUs.**cpu\_time\_ns (optional)***Integer* The CPU time of the instance.**4. InstanceNetworkInterface****interface\_id (optional)***Long* The ID of the network interface based on the DB. format: int64**network\_id (optional)***Long* The ID of the network based on the DB. format: int64**name (optional)***String* The name of the interface.**mac (optional)***String* The MAC of the interface.**ip (optional)***String* The IP of the instance.**5. ListNodes****nodes (optional)***array[String]***6. NbdPort****port (optional)***BigDecimal* format: int32**7. Network****id (optional)***Long* format: int64**name (optional)***String***virtual (optional)***Boolean* If the network is virtual or not.**vnet\_id (optional)***String* The network ID from the virtxd.**resource\_group\_ids (optional)***array[Long]* The list of resource group IDs that use the network. format: int64**ip\_start (optional)***String* The start of the IP range.**ip\_end (optional)***String* The end of the IP range.**cidr (optional)**

String The CIDR of the network.

**gateway (optional)**

String The gateway of the network.

**dhcp\_server\_ip (optional)**

String The dhcp server of the network. It is the ip of the network interface of the dhcp namespace and needs to be specified only if the dhcp\_type='internal' and the routable='true'.

**nat\_forwarding (optional)**

Boolean It translates traffic from the current network to the external network and needs to be specified only if the dhcp\_type='internal' and the routable='true'

**ip\_version (optional)**

Integer

**dhcp\_type (optional)**

String

The type of the DHCP

- 'internal' means that the API will serve IPs to the instances, from an internal DHCP server.
- 'static' means that the API will write static IPs to the VMs when provisioning them.
- 'external' means that the API will not serve any IP to the instances.

**Enum:**

internal

static

external

**is\_routable (optional)**

Boolean It will enable a tunnel for the virtual network when it is true.

**active\_nodes (optional)**

array[String] It show all the NexVisor's IDs that have enabled the network.

**physical\_paths (optional)**

array[Long] The paths that are used, for the specific network, to communicate between the NexVisors.  
format: int64

**is\_private (optional)**

Boolean A private network is created when a new cluster is created.

**is\_vlan (optional)**

Boolean VLAN is new kind of network, which is a virtual network that does not need to transmit the packets based on the MAC address but the specified ID of a VIF ( Virtual InterFace). Enable only the new networks to be VLAN. Can not edit the attribute.

**is\_wire\_vlan (optional)**

Boolean If VLAN is enabled, then the user need to decide if he/she needs the VLAN to be wired.

**vlan\_tag (optional)**

Long If VLAN is enabled, then the user will have to add a tag for the VLAN. The tag can not be a number between 0-4095. format: int64

**is\_dedicated (optional)**

Boolean Only the network is physical, then it will return this attribute. If it is 'true', then it means the physical network will be not used as a physical path for a virtual network.

**mac (optional)**

String Only the physical networks will return this attribute. This attribute shows the MAC of the physical port that the physical network uses.

## 8. NewInstance

**name (optional)**

String

**node\_id (optional)**

String

**vcpus (optional)**

BigDecimal format: int32

**ram (optional)**

BigDecimal format: int32

**configdriver (optional)**

String

**ips (optional)**

array[NewInstanceIP]

**volumes (optional)**

array[NewInstanceVolume]

## 9. NewInstanceAction

**action (optional)**

String

**Enum:**

start

stop

reboot

migrate

pause

unpause

**node\_id (optional)**

String This attribute is needed only for the action 'migrate'. It represents in which node to migrate the instance.

**restart\_after (optional)**

Boolean This attribute is needed only for the action 'migrate'. It will say if the instance will restart after the migration.

## 10. NewInstanceIP

**address (optional)**

String

**mac (optional)**

String

## 11. NewInstanceVolume

**name (optional)**

String

**delete\_on\_termination (optional)**

Boolean

## 12. NewVolume

**name (optional)**

String



**size (optional)**

BigDecimal The size of the volume in GB format: int64

**13. Resources****vcpus (optional)**

BigDecimal format: int64

**memory\_mb (optional)**

BigDecimal format: int64

**local\_gb (optional)**

BigDecimal format: int64

**vcpus\_used (optional)**

BigDecimal format: int64

**memory\_mb\_used (optional)**

BigDecimal format: int64

**local\_gb\_used (optional)**

BigDecimal format: int64

**disk\_available\_least (optional)**

BigDecimal format: int64

**14. Volume**

This model represents a virtual disk.

**id (optional)**

Long format: int64

**name (optional)**

String

**instance\_id (optional)**

Long The instance's ID that the volume is attached to. format: int64

**size (optional)**

Long The size of the volume. format: int64

**status (optional)**

String The status of the volume.

**Enum:**

in-use

error

available

downloading

**disks (optional)**

array[String] This array holds all the ids related to the physical disks that the volume utilizes. By definition, these disks all belong to the same datastore.

**datastore\_id (optional)**

String The ID of the datastore.

**vdisk\_id (optional)**

String The ID of the volume from the onappstore.

**is\_degraded (optional)**

Boolean Shows if the volume is degraded and needs repairs.

**is\_repairing (optional)**

Boolean Shows if the OpenStack started repairing the volume.

**percent\_repair (optional)**

*Integer* It shows how much of the volume is repaired in percentage. It shows only when the 'is\_repairing' is true. format: int32

### 6.3 Hypervisor Performance counters

This section includes the current list of performance counters provided by the MicroVisor and the reported output over the Ethernet (eth1 interface) from a run on a generic x86 platform (Supermicro server with Intel XeonCPUs). Please note that for brevity we have shortened the output, showing only the first 4 CPUs and removing spaces.

```
# ./mvctl-perfc -i eth1 --mvmac a0369f2058c4 --perfc-dump
Result:
Performance counters SHOW (now = 0x00000023:BC84B9EE)
exceptions          TOTAL[ 408227] CPU00[ 316170] CPU01[ 92057] CPU02[0]
vmexits             TOTAL[      0]
cause vector        TOTAL[      0]
SVMexits            TOTAL[      0]
segmentation fixups TOTAL[      0]
apic timer interrupts
TOTAL[ 151139] CPU00[ 143985] CPU01[ 2251] CPU02[ 2] CPU03[ 4]...
domain page tlb flushes
TOTAL[ 5566] CPU00[ 3226] CPU01[ 2340] CPU02[ 0] CPU03[ 0] ...
calls to mmuext_op
TOTAL[ 114921] CPU00[ 96998] CPU01[ 17923] CPU02[ 0] CPU03[ 0] ...
mmuext ops
TOTAL[ 119460] CPU00[ 99464] CPU01[ 19996] CPU02[ 0] CPU03[ 0]
calls to mmu_update
TOTAL[ 497812] CPU00[ 304835] CPU01[192977] CPU02[ 0] CPU03[ 0]
page updates
TOTAL[ 539464] CPU00[ 327210] CPU01[ 212254] CPU02[ 0] CPU03[ 0]
mmu_updates of writable pages
TOTAL[ 43730] CPU00[ 23568] CPU01[ 20162] CPU02[ 0] CPU03[ 0]
calls to update_va_map
TOTAL[ 70865] CPU00[ 53023] CPU01[ 17842] CPU02[ 0] CPU03[ 0]...
page faults
TOTAL[ 146935] CPU00[ 76657] CPU01[ 70278] CPU02[ 0] CPU03[ 0]...
copy_user faults
TOTAL[ 12] CPU00[ 12] CPU01[ 0] CPU02[ 0] CPU03[ 0]...
map_domain_page count
TOTAL[ 673094] CPU00[ 420994] CPU01[ 252100] CPU02[ 0] CPU03[ 0]...
writable pt emulations
TOTAL[ 74788] CPU00[ 41081] CPU01[ 33707] CPU02[ 0] CPU03[ 0]...
pre-exception fixed TOTAL[      0]
guest pagetable walks TOTAL[      0]
calls to shadow_alloc TOTAL[      0]
shadow_alloc flushed TLBs TOTAL[      0]
```

number of shadow pages in use	TOTAL[	0]
calls to shadow_free	TOTAL[	0]
shadow recycles old shadows	TOTAL[	0]
shadow recycles in-use shadows	TOTAL[	0]
shadow hit read-only linear map	TOTAL[	0]
shadow A bit update	TOTAL[	0]
shadow A&D bit update	TOTAL[	0]
calls to shadow_fault	TOTAL[	0]
shadow_fault fast path n/p	TOTAL[	0]
shadow_fault fast path mmio	TOTAL[	0]
shadow_fault fast path error	TOTAL[	0]
shadow_fault guest bad gfn	TOTAL[	0]
shadow_fault really guest fault	TOTAL[	0]
shadow_fault emulates a read	TOTAL[	0]
shadow_fault emulates a write	TOTAL[	0]
shadow_fault emulator fails	TOTAL[	0]
shadow_fault emulate stack write	TOTAL[	0]
shadow_fault emulate for CR0.WP=0	TOTAL[	0]
shadow_fault fast emulate	TOTAL[	0]
shadow_fault fast emulate failed	TOTAL[	0]
shadow_fault handled as mmio	TOTAL[	0]
shadow_fault fixed fault	TOTAL[	0]
shadow causes ptwr to emulate	TOTAL[	0]
calls to shadow_validate_g11e	TOTAL[	0]
calls to shadow_validate_g12e	TOTAL[	0]
calls to shadow_validate_g13e	TOTAL[	0]
calls to shadow_validate_g14e	TOTAL[	0]
calls to shadow_hash_lookup	TOTAL[	0]
shadow hash hit in bucket head	TOTAL[	0]
shadow hash misses	TOTAL[	0]
calls to get_shadow_status	TOTAL[	0]
calls to shadow_hash_insert	TOTAL[	0]
calls to shadow_hash_delete	TOTAL[	0]
shadow removes write access	TOTAL[	0]
shadow writeable: 32b w2k3	TOTAL[	0]
shadow writeable: 32pae w2k3	TOTAL[	0]
shadow writeable: 64b w2k3	TOTAL[	0]
shadow writeable: linux low/solaris	TOTAL[	0]
shadow writeable: linux high	TOTAL[	0]
shadow writeable: FreeBSD	TOTAL[	0]
shadow writeable: s11p	TOTAL[	0]
shadow writeable: s11p failed	TOTAL[	0]
shadow writeable brute-force	TOTAL[	0]
shadow writeable resync bf	TOTAL[	0]

shadow removes all mappings	TOTAL[	0]
shadow rm-mappings brute-force	TOTAL[	0]
shadow unshadows for fork/exit	TOTAL[	0]
shadow unshadows a page	TOTAL[	0]
shadow unshadow by up-pointer	TOTAL[	0]
shadow unshadow brute-force	TOTAL[	0]
shadow_get_page_from_l1e failed	TOTAL[	0]
shadow checks gwalk	TOTAL[	0]
shadow check inconsistent gwalk	TOTAL[	0]
shadow flush tlb by removing write perm	TOTAL[	0]
shadow emulates invlpg	TOTAL[	0]
shadow invlpg faults	TOTAL[	0]
shadow extra pt write	TOTAL[	0]
shadow extra non-pt-write op	TOTAL[	0]
shadow extra emulation failed	TOTAL[	0]
shadow OOS fixup adds	TOTAL[	0]
shadow OOS fixup evictions	TOTAL[	0]
shadow OOS unsyncs	TOTAL[	0]
shadow OOS evictions	TOTAL[	0]
shadow OOS resyncs	TOTAL[	0]
MS Hv Switch Address Space	TOTAL[	0]
MS Hv Flush TLB list	TOTAL[	0]
MS Hv Flush TLB all	TOTAL[	0]
MS Hv Notify long wait	TOTAL[	0]
MS Hv Flush TLB	TOTAL[	0]
MS Hv rdmsr Guest OS ID	TOTAL[	0]
MS Hv rdmsr hypercall page	TOTAL[	0]
MS Hv rdmsr vp index	TOTAL[	0]
MS Hv rdmsr TSC frequency	TOTAL[	0]
MS Hv rdmsr APIC frequency	TOTAL[	0]
MS Hv rdmsr time ref count	TOTAL[	0]
MS Hv rdmsr icr	TOTAL[	0]
MS Hv rdmsr tpr	TOTAL[	0]
MS Hv rdmsr APIC assist	TOTAL[	0]
MS Hv rdmsr APIC msr	TOTAL[	0]
MS Hv rdmsr TSC msr	TOTAL[	0]
MS Hv wrmsr Guest OS ID	TOTAL[	0]
MS Hv wrmsr hypercall page	TOTAL[	0]
MS Hv wrmsr vp index	TOTAL[	0]
MS Hv wrmsr icr	TOTAL[	0]
MS Hv wrmsr tpr	TOTAL[	0]
MS Hv wrmsr eoi	TOTAL[	0]
MS Hv wrmsr APIC assist	TOTAL[	0]
MS Hv wrmsr APIC msr	TOTAL[	0]

```

MS Hv wrmsr TSC msr          TOTAL[          0]
realmode instructions emulated TOTAL[          0]
vmexits from realmode        TOTAL[          0]
vmexits from Pause-Loop Detection TOTAL[          0]
hypercalls
TOTAL[ 889810] CPU00[ 549614] CPU01[ 340196] CPU02[ 0] CPU03[ 0]...
calls to multicall
TOTAL[ 11538] CPU00[ 6505] CPU01[ 5033] CPU02[ 0] CPU03[ 0]...
calls from multicall
TOTAL[ 83333] CPU00[ 59349] CPU01[ 23984] CPU02[ 0] CPU03[ 0]...
#interrupts
TOTAL[ 190171] CPU00[ 150057] CPU01[ 9435] CPU02[ 2609] CPU03[ 2680]...
#IPIs
TOTAL[ 37697] CPU00[ 5219] CPU01[ 7184] CPU02[ 2533] CPU03[ 2514]...
sched: timer
TOTAL[ 116] CPU00[ 68] CPU01[ 48] CPU02[ 0] CPU03[ 0]...
sched: runs through scheduler
TOTAL[ 13992] CPU00[ 6666] CPU01[ 6361] CPU02[ 149] CPU03[ 325]...
sched: context switches
TOTAL[ 12640] CPU00[ 6332] CPU01[ 6308] CPU02[ 0] CPU03[ 0]...
sched: specific scheduler
TOTAL[ 13992] CPU00[ 6666] CPU01[ 6361] CPU02[ 149] CPU03[ 325]...
sched: dom_init
TOTAL[ 2] CPU00[ 2] CPU01[ 0] CPU02[ 0] CPU03[ 0]...
sched: vcpu_alloc
TOTAL[ 18] CPU00[ 18] CPU01[ 0] CPU02[ 0] CPU03[ 0]...
sched: vcpu_insert
TOTAL[ 2] CPU00[ 2] CPU01[ 0] CPU02[ 0] CPU03[ 0]...
sched: vcpu_remove          TOTAL[ 0]
sched: vcpu_sleep
TOTAL[ 5] CPU00[ 3] CPU01[ 2] CPU02[ 0] CPU03[ 0]...
sched: vcpu_yield
TOTAL[ 120] CPU00[ 120] CPU01[ 0] CPU02[ 0] CPU03[ 0]...
sched: vcpu_wake_running
TOTAL[ 11] CPU00[ 6] CPU01[ 5] CPU02[ 0] CPU03[ 0]...
sched: vcpu_wake_onrunq          TOTAL[ 0]
sched: vcpu_wake_runnable
TOTAL[ 6317] CPU00[ 3868] CPU01[ 2430] CPU02[ 0] CPU03[ 9]...
sched: vcpu_wake_not_runnable TOTAL[ 0]
sched: tickled_no_cpu          TOTAL[ 0]
sched: tickled_idle_cpu
TOTAL[ 6317] CPU00[ 3868] CPU01[ 2430] CPU02[ 0] CPU03[ 9]...
sched: tickled_busy_cpu          TOTAL[ 0]
sched: vcpu_check

```

TOTAL[	27984]	CPU00[	13332]	CPU01[	12722]	CPU02[	298]	CPU03[	650]...
csched: delay									
TOTAL[	11]	CPU00[	7]	CPU01[	4]	CPU02[	0]	CPU03[	0]...
csched: acct_run									
TOTAL[	179]	CPU00[	179]	CPU01[	0]	CPU02[	0]	CPU03[	0]...
csched: acct_no_work									
TOTAL[	9586]	CPU00[	4695]	CPU01[	1]	CPU02[	0]	CPU03[	0]...
csched: acct_balance									
TOTAL[	18]	CPU00[	18]	CPU01[	0]	CPU02[	0]	CPU03[	0]...
csched: acct_reorder									
csched: acct_min_credit			TOTAL[		0]				
csched: acct_vcpu_active									
TOTAL[	80]	CPU00[	42]	CPU01[	38]	CPU02[	0]	CPU03[	0]...
csched: acct_vcpu_idle									
TOTAL[	80]	CPU00[	80]	CPU01[	0]	CPU02[	0]	CPU03[	0]...
csched: vcpu_boost									
TOTAL[	252]	CPU00[	146]	CPU01[	106]	CPU02[	0]	CPU03[	0]...
csched: vcpu_park									
csched: vcpu_unpark			TOTAL[		0]				
csched: load_balance_idle									
TOTAL[	6625]	CPU00[	3232]	CPU01[	3155]	CPU02[	75]	CPU03[	163]...
csched: load_balance_over									
TOTAL[	184]	CPU00[	184]	CPU01[	0]	CPU02[	0]	CPU03[	0]...
csched: load_balance_other									
csched: steal_trylock			TOTAL[		0]				
csched: steal_trylock_failed			TOTAL[		0]				
csched: steal_peer_idle			TOTAL[		0]				
csched: migrate_queued			TOTAL[		0]				
csched: migrate_running			TOTAL[		0]				
csched: migrate_kicked_away			TOTAL[		0]				
csched: vcpu_hot			TOTAL[		0]				
csched2: burn_credits_t2c			TOTAL[		0]				
csched2: acct_load_balance			TOTAL[		0]				
csched2: update_max_weight_quick			TOTAL[		0]				
csched2: update_max_weight_full			TOTAL[		0]				
csched2: migrate_requested			TOTAL[		0]				
csched2: migrate_on_runq			TOTAL[		0]				
csched2: migrate_no_runq			TOTAL[		0]				
csched2: runtime_min_timer			TOTAL[		0]				
csched2: runtime_max_timer			TOTAL[		0]				
csched2: migrated			TOTAL[		0]				
csched2: migrate_resisted			TOTAL[		0]				
csched2: credit_reset			TOTAL[		0]				
csched2: deferred to tickled cpu			TOTAL[		0]				

```
csched2: tickled_cpu_overwritten  TOTAL[          0]
csched2: tickled_cpu_overridden   TOTAL[          0]
```

```
PG_need_flush tlb flushes
```

```
TOTAL[      28177] CPU00[      15613] CPU01[ 12564] CPU02[    0] CPU03[    0]...
```